MASTER EN SCIENCES INFORMATIQUES

Rétroingénieurie et développement d'une interface d'acquisition de connaissances

Wauthier, Sandy

Award date:
2010

Awarding institution:
Universite de Namur

Rétroingénieurie et développement d'une interface d'acquisition de connaissances

Master 60 - Faculté d'informatique

Facultés universitaires Notre Dame de la Paix

Sandy Wauthier

Année académique 2009 - 2010

Résumé

En Europe, le secteur de la métallurgie est confronté à de multiples contraintes tant sur le plan des fonctionnalités offertes que de nouvelles normes écologiques à respecter. Soucieuse de la prospérité de sa sidérurgie, la Région Wallonne a voulu l'aider à concilier ces nouvelles contraintes en finançant un projet de recherche visant à développer un système expert en planification de traitement de matériaux. Il est actuellement à l'état de prototype et l'une de ses interfaces, l'interface d'acquisition de connaissances, nécessite certaines amélioration. Afin de les rencontrer, nous avons effectué une rétroingénierie de cette interface et avons ainsi mis en évidence ses faiblesses. En outre, nous avons consulté un spécialiste du domaine des matériaux pour procéder à une analyse des exigences et dégager de nouvelles fonctionnalités à incorporer. En nous basant sur ces deux étapes, nous avons enfin développé une toute nouvelle interface utilisant, entre autres, les technologies Flex et Zend-AMF.

Abstract

The metallurgy sector in Europe is confronted to multiple constraints involving both functionnal and ecological requirements. The Walloon Region is particularly concerned with this sector's prosperity. In an effort to concile these various requirements, the region invested in a research project. Its purpose was to develop an expert system capable of planifying sequences of treatments on materials. The development is currently a prototype and one of its user interfaces, the user interface for acquiring new knowledge, needs to be improved. To be able to make the improvements, we reverse-engineered the interface and brought to light its weaknesses. We met with a specialist in metallurgy to put together a requirements analysis wich brought forth new functionnalities to integrate. Based on these two steps, we finally developped a brand new interface using Flex and Zend-AMF technologies.

Remerciements

En premier lieu, je souhaite remercier mon promoteur, M Jacquet, qui m'a confié ce sujet de mémoire et a suivi avec attention mon avancement tout au long de sa réalisation.

Je tiens à remercier également M Bertrand, spécialiste du domaine des matériaux, pour son aide et sa disponibilité. Ses conseils et ses connaissances du domaine d'application ont été d'une aide précieuse.

Enfin, je réserve un grand merci à Mathieu Detollenaere pour ses conseils, sa patience, et sa relecture du mémoire.

Préface

Le monde de la métallurgie en Europe est confronté à des contraintes que ne recontrent pas nécessairement ses concurrents étrangers. En particulier, l'Union Européenne a établi de nouvelles normes écologiques qui complexifient considérablement la tâche des concepteurs industriels. Un traitement de surface permet de conférer à un matériau donné un nouvel ensemble de propriétés. Les nouvelles contraintes écologiques accroissent considérablement les propriétés exigées tout en limitant les traitements possibles. Un seul traitement ne suffisant pas toujours à conférer l'ensemble de propriétés demandées, il faut recourir à des solutions en plusieurs couches. Le développement de telles solutions exige une connaissance exhaustive des différents matériaux et procédés disponibles. Cela constitue un réel obstacle pour les plus petits acteurs du domaine. Consciente de ces nouvelles difficultés, la Région Wallonne a financé un projet de recherche impliquant l'Université libre de Bruxelles et les Facultés Universitaires Notre Dame de la Paix de Namur afin de développer un système informatique capable d'élaborer de telles solutions. Il s'agit du système expert *Expesurf*.

L'application comporte différents composants dont une interface permettant de compléter de nouvelles données l'ensemble des connaissances du système. Cette interface permet également de visualiser et parcourir les connaissances déjà acquises. Elle pose actuellement quelques problèmes liés, d'une part, à la technologie dans laquelle elle a été développée et, d'autre part, à l'utilisation de certaines de ses fonctionnalités. Ce mémoire vise à effectuer une rétroingénieurie de l'interface existante afin de définir précisément ses différents manquements pour ensuite proposer une solution corrective.

Nous effectuerons dans un premier temps une rétroingénieurie de l'existant. Le fonctionnement de l'interface est indissociable de la base de données, au coeur du logiciel. Une première étape de cette rétroingénieurie analysera donc cette base de données. Certaines fonctionnalités spécifiques à l'interface d'acquisition de connaissances ont été développées à même la base de données. Celle-ci présente, par ailleurs, certains éléments structurels dont l'intérêt a été remis en question. On comprendra au travers de cette partie qu'afin de faciliter les encodages et de remanier l'interface, on ne pourra se contenter d'utiliser la base de données dans son état actuel.

L'interface a été développée dans une technologie particulière : le framework Zope. Lors du développement initial cette technologie semblait promise à un bel avenir qui ne s'est malheureusement pas avéré. En effet, la maintenance de cette version du framework a été abandonnée. Aujourd'hui certains dysfonctionnements de l'interface découlent directement de ce choix technologique qu'il convient donc de comprendre et sur lequel il est intéressant de poser un regard critique. Notre rétroingénieurie se penchera donc ensuite sur la technologie de l'application et l'architecture de son développement.

Finalement, d'autres faiblesses indépendantes de la technologie employée, découlent directement du choix des fonctionnalités et de la manière dont elles ont été développées. En effet, l'organisation de l'interface ne permet pas de présenter certains types d'informations et rend l'encodage de nouvelles données fastidieux.

En conclusion, les manquements fonctionnels de l'interface auraient pu être solutionnés au travers d'une simple correction de l'interface existante. En revanche, les problèmes soulevés par le développement en Zope ne sont pas corrigibles. Nous avons donc établi qu'il faudrait développer une toute nouvelle interface. Nous avons tenté de préserver les qualités de l'interface existante tout en corrigeant ses lacunes.

En plus de ces corrections, nous avons donné l'opportunité à l'utilisateur de formuler de nouvelles attentes apparues avec le temps et au travers d'une utilisation régulière de l'application. Cette opportunité a débouché sur une analyse complète des exigences que nous décrirons dans une seconde partie. Cette analyse a permis de construire avec le client les prototypes de la nouvelle interface et de définir, avec précision, les fonctionnalités et la manière dont il souhaite y accéder.

Enfin, la rétroingénieurie et l'analyse des exigences aidant, nous avons pu entammer le développement de notre nouvelle interface. La technologie du développement initial posant problème, nous avons dû nous pencher sur de nouvelles solutions. Les exigences du client ont permis d'isoler un ensemble de technologies susceptibles de convenir au développement d'une interface dynamique et visuellement attractive. La critique de la technologie employée initialement a permis ensuite de trier parmi les technologies candidates celles qui ne semblent pas tomber dans les mêmes travers ni risquer le même sort que Zope. Notre choix s'est arrêté sur Flex dont nous découvrirons le fonctionnement au travers d'exemples choisis.

Un bon choix technologique facilite et accélère le développement de l'application mais il ne dispense pas d'une architecture soignée garante de modularité et de lisibilité du programme. Celle-ci permettra à de futurs développeurs de facilement s'approprier le code de l'application afin de le maintenir et de

l'améliorer. Cette architecture est à cheval sur deux composants principaux : le client et le serveur. Il incombe au développeur de découper son application de façon modulaire afin que des parties utiles de l'application soient facilement réutilisables. C'est le premier critère de découpe des deux composants.

La construction de l'architecture du client a été guidée plus particulièrement par le pattern «modèle, vue, controlleur» que nous avons veillé à respecter. Le reste de l'architecture a été découpée afin de veiller à garder une cohérence intuitive par rapport à l'interface que l'on a sous les yeux et donc l'agencement dépend déjà d'une découpe logique et fonctionnelle qu'il est intéressant de retrouver dans le code de l'application. La découpe globale de l'application se fera donc en fonction des groupements de procédures et d'élements d'interfaces qui permettent de proposer une certaine fonctionnalité.

Cette architecture rend compte du squelette de la nouvelle interface mais certains éléments plus techniques de l'implémentation permettent d'être présentés. En effet, des choix de développement plus critiques ont été fait afin de rencontrer les objectifs de l'interface sans altérer significativement la base de données, ou faciliter des futurs développements. Ces points stratégiques seront explorés plus en détail avant de clôturer la description du développement avec une présentation de l'interface développée et de tests dont elle a fait l'objet.

Enfin, à la fin de l'analyse des exigences, les grandes attentes de l'application et la manière dont on souhaite les voir matérialisées sont présentées mais la phase de développement et, en particulier, la phase de test sont l'occasion de se rendre compte de toutes sortes de fonctionnalités qu'on pourrait vouloir retrouver dans l'application.

Au travers de la rédaction de ce mémoire nous allons chercher à rencontrer trois objectifs. Le premier objectif est de proposer une rétroingénieurie suffisamment complète que pour qu'elle puisse servir de documentation à tout futur développement. En effet, la rétro-ingénieurie fait l'analyse de la base de données donc dépendent tous les autres composants du système. Un second problème consiste à construire une interface adaptée aux besoins des utilisateurs d'Expesurf. Elle devra rencontrer les attentes formulées d'emblée par le client mais, également, trouver des solutions créatives aux problèmes pour lesquels il ne formule pas d'attente spécifique. Il nous faudra enfin nous assurer que des développement futurs soient facilités par les choix technologiques, d'implémentation, et architecturaux effectués dans le cadre de ce mémoire. En partculier, ce document devra constituer une documentation précise et utile de la nouvelle interface.

Table des matières

Chapitre 1

Introduction

1.1 Mise en contexte

Le secteur de la métallurgie en Europe doit faire face à de nouveaux challenges depuis peu. De nouvelles normes européennes exigent que le secteur adapte rapidement certaines méthodes de travail pour rencontrer des objectifs écologiques. De plus, le marché s'étend et voit apparaître de nouveaux acteurs concurrents. Ces nouvelles contraintes vont à l'encontre d'un secteur où le changement se fait très progressivement. En effet, on ne peut évoluer en métallurgie avec un système d'essais et d'erreurs car les budgets de développement sont systématiquement énormes et les conséquences des erreurs, trop importantes. De plus, les interactions entre matériaux sont telles qu'il est titanesque d'envisager toutes les combinaisons de procédés susceptibles de mener à une solution. De ce fait, les choix sont toujours dictés par l'expérience du métier et laissent peu de place pour une révision totale des procédés et traitements.

Il existe néanmoins un facteur qui a été à la source de la plupart des progrès technologiques de ces dernières décennies : les traitements de surfaces. Ils interviennent dans tous les domaines de technologie de pointe et leur utilisation s'est fortement accrue ces dernières années. La difficulté n'est pas de trouver le traitement d'une surface qui va lui permettre d'acquérir une propriété donnée : ces informations-la sont déjà librement accessibles. Il s'agit plutôt de trouver un ensemble de couches successives qui va permettre de conférer à un matériau donné une propriété impossible à obtenir au moyen d'une seule couche ou traitement. Ces techniques ne sont actuellement accessibles qu'à un nombre restreint d'industriels qui ont les moyens de rassembler toutes les informations nécessaires.

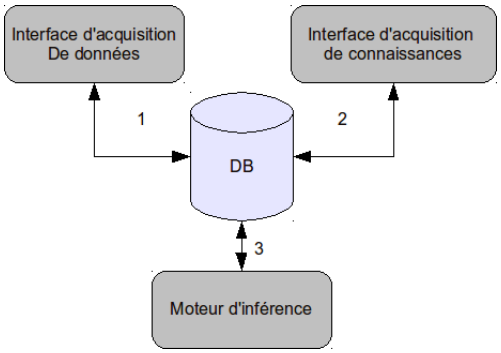


Fig. 1.1 – Expesurf

Cette application est composée de quatre composants principaux : une interface d'acquisition de données, une interface d'acquisition de connaissances, une base de données et un moteur d'inférence. Ils procurent une intelligence artificielle (voir figure 1.1) au système en élaborant des solutions sur base d'un ensemble de connaissances établies dans la base de données. L'interface d'acquisition de connaissances permet d'enrichir le savoir du système. L'interface d'acquisition de données permet de soumettre un problème et d'aller récupérer les solutions avec la démonstration de leur validité (1). La base de données joue un double rôle. Elle contient toutes les données du domaine métallurgique ainsi que toutes

les informations qui transitent entre les interfaces et le moteur d'inférence. Le moteur d'inférence fonctionne de manière régulière mais ne renvoie pas les résultats en temps réel : on dépose un problème, et on revient à postériori vérifier s'il a été solutionné. Le moteur d'inférence ne travaille pas directement sur les informations de la base de données. Il les traduit d'abord dans un format qu'il sait manipuler. Il se synchronise donc régulièrement avec les informations, et problèmes enregistrés en base de données(3).

Le cycle de vie de ce logiciel est pour l'instant arrêté avant la première phase de test. La phase de test ne peut s'effectuer qu'avec une garantie formelle de la validité des solutions proposées et donc des raisonnements effectués par le logiciel d'une part, et des prémisses sur lesquelles il se base (ensemble des données du logiciel) d'autre part. Or pour l'instant certaines solutions semblent inadéquates. L'interface d'acquisition de connaissances proposée n'est pas adaptée aux encodages nécessaires et ne permet pas une visualisation satisfaisante des informations. Toutefois, ce qui fait la qualité des réponses proposées par le logiciel, c'est d'un côté la puissance de l'intelligence artificielle qui effectue la recherche mais également toutes les connaissances sur lesquelles se basent son raisonnement. Une meilleure lisibilité des données au travers de l'interface et un encodage plus clairs permettraient de comprendre puis résoudre les défauts possibles du moteur d'inférence, de compléter de manière plus appropriée les données de la base de données, et *in fine* d'entamer la phase de test. Ce mémoire vise à redévelopper l'interface d'acquisition de connaissances.

1.2 Présentation de l'interface existante

Cette interface permet d'enrichir la base de donnaissances du système expert en définitive, en quoi consistent ces connaissances ? En effet, pour mieux comprendre l'interface il est essentiel de cerner les différents types de connaissances que l'on va retrouver en base de données et le type de solution que va apporter le système expert dans son ensemble[1].

Le problème que résoude le système expert est de partir d'une pièce composée d'un matériau donné avec un ensemble de propriétés initiales et d'appliquer d'autres couches de matériaux et des procédés à cette pièce de manière à ce qu'elle en resorte avec un nouvel ensemble de propriétés.

On comprend rapidement qu'au coeur de ce domaine de connaissances on retrouve des matériaux. Les matériaux peuvent être des métaux, des céramiques etc. Ils sont organisés en familles : l'acier fait partie de la famille des métaux.

Le système va devoir en connaître les différentes propriétés de chaque matériau. Ces propriétés reprennent des propriétés aussi simples que «de couleur rouge» ou des propriétés thermiques, magnétiques, de conductivité électrique...

Ensuite, le système va devoir connaitre des ensembles de procédés et sur quel type de pièces et matériaux ils peuvent être appliqués. Ces procédés peuvent reprendre Finalement, il va devoir prendre en compte les incompatibilités qui peuvent intervenir entre les matériaux et certains procédés.

L'interface d'acquisition de connaissances vise donc à insérer des matériaux, leurs propriétés, des procédés, et les incompatibilités excluant certaines combinaisons de matériaux et procédés. Le système partira ainsi de la description de la pièce et de son matériau pour progressivement se rapprocher de l'ensemble de propriétés que l'on souhaite en appliquant successivement un procédé après l'autre pour obtenir une succession de traitements qui permettent à la pièce de remplir l'ensemble des propriétés désirées.

Cette interface est également une fenêtre sur la base de connaissances. Elle est actuellement le seul moyen donné à l'utilisateur de l'application pour découvrir l'ensemble des connaissances déjà encodées.

1.2.1 Menus et page d'accueil

La première restriction notable du logiciel est la contrainte de s'en servir avec le navigateur Mozilla Firefox. En effet, le logiciel n'est pas adapté à une utilisation sous Internet Explorer où certaines parties des formulaires (affichage dynamique des nouveaux champs au fur et à mesure de la complétion) ne fonctionnent pas.

[1] Notez que cette brève présentation vise uniquement à faciliter la compréhension de l'interface mais pas de comprendre le domaine d'application qui est bien trop complexe que pour pouvoir être décrit dans tout ses détails.

L'application présente une première interface de login basique demandant à l'utilisateur de s'identifier avant de présenter la page ci-dessous. Cette page est la page de validation des données. Tout nouvel encodage, suppression, ou édition de donnée par une personne doit être validée par une personne tierce afin que la modification soit définitivement apportée à la base de connaissances. Cette fonctionnalité faisait partie des exigences requises par les utilisateurs du système mais s'avère contre-productive et contraignante au fil de son utilisation.

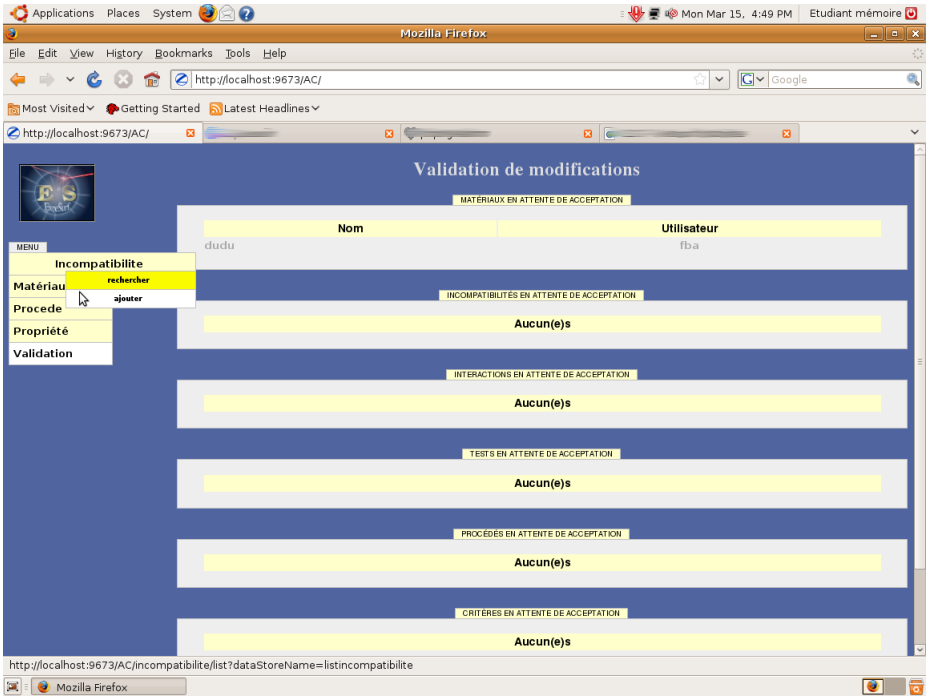


FIG. 1.2 – Page d'accueil et menu

Le menu est un menu latéral gauche qui divise les tâches en fonction du type d'objet auquel elles se rapportent (voir figure 1.2). On peut ainsi effectuer une recherche et un ajout pour chaque entité du menu. Parfois un menu mène à une recherche ou un ajout d'un type d'entité connexe (comme dans le cadre de la section matériaux qui permet les opérations sur les interactions également).

1.2.2 Encodage de données

L'encodage des données se fait au moyen de formulaires. Ces formulaires sont communs à l'édition (auquel cas ils sont préremplis avec les données connues) ou à l'insertion comme c'est le cas ici pour les matériaux (voir figure 1.3).

Certains des formulaires sont particulièrement simples et courts (propriétés) tandis que d'autres imposent un encodage fastidieux. C'est le cas de l'encodage des interactions par exemple. Cet encodage intialement clair et simple demande qu'on encode en parallèle les solutions, ce qui accroît considérablement la longueur du formulaire initial (voir figure 1.4).

Cette tâche est rendue d'autant plus ardue qu'elle réclame un encodage hiérarchique du matériau que l'on va classer en fonction de sa famille, sous famille etc. A chaque parent encodé, un nouveau champ se déploie et réclame l'encodage du parent suivant et ainsi de suite (voir figure 1.5). Se cachent donc derrière certaines lignes du formulaire jusqu'à six encodages.

Cet encodage est inévitable. Il répond à la structure familiale des matériaux et propriétés et prévoyait de faciliter la recherche et de ne pas avoir à réencoder les propriétés d'une famille pour chacun de ses enfants.

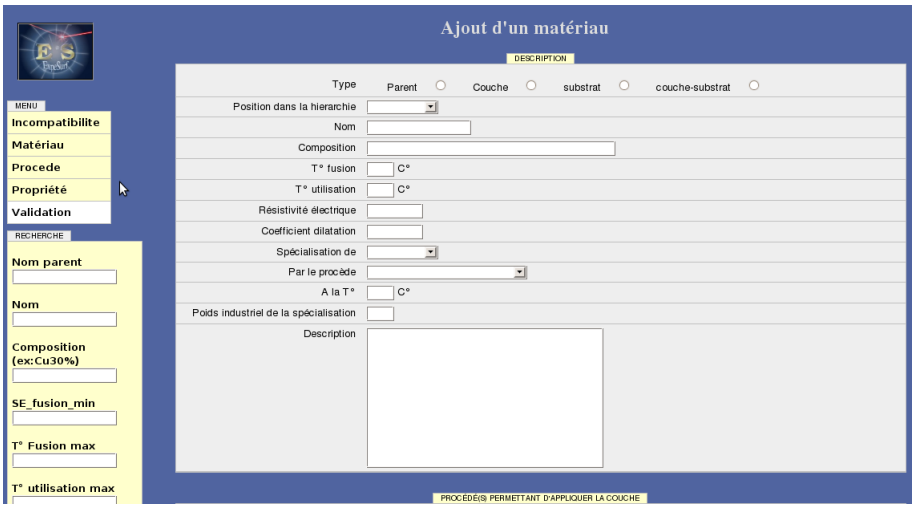


FIG. 1.3 – Encodage des matériaux

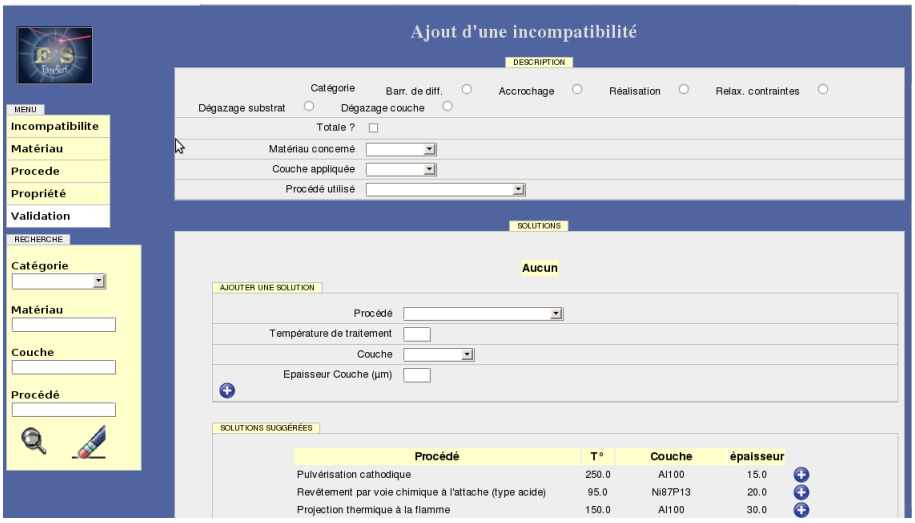


FIG. 1.4 – Composants de Zope

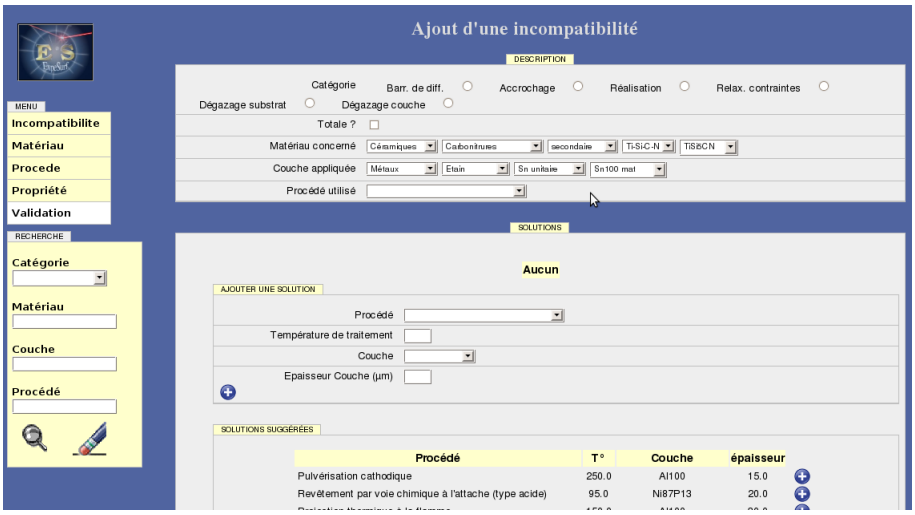


FIG. 1.5 – Démultiplication des champs

FIG. 1.6 – Naviguer dans les données

1.2.3 Recherche de données

La recherche est possible selon deux modes différents : un mode navigationnel et une recherche spécifique au travers d'un formulaire. Le formulaire varie d'une page à l'autre en fonction des champs spécifiques à l'objet de la page mais est présent qu'il s'agisse d'une page d'encodage de données ou de navigation. La navigation présente la structure des données en fonction de l'organisation hiérarchique et familiale des informations (voir figure 1.6). On peut encoder une demande de suppression ou d'édition de données pour chaque type de donnée. Cette demande fera l'objet d'une validation en page d'accueil pour toute personne qui fait utilisation du logiciel.

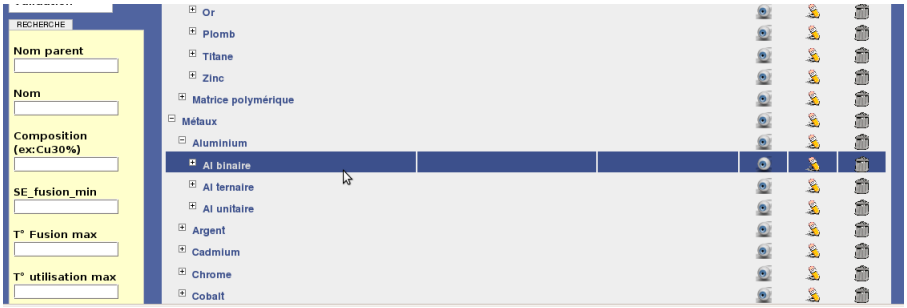


FIG. 1.7 – Structure familiale des données

Le principal défaut de la recherche via formulaire est que l'ensemble des champs de la recherche ne comprend pas toujours toutes les informations d'intérêt (toutes les différentes nomenclatures pour les matériaux ne sont pas reprises par exemple) mais également que ne seront repris dans les résulats que les éléments de recherche qui répondent exactement aux termes de la recherche (une recherche du mot "Cuivre" ne renverra pas tous les cuivres mais uniquement le matériau nommé "Cuivre"). La recherche navigationnelle de son côté impose de circuler parmi les familles (voir figure 1.7) ce qui oblige à déployer jusqu'à cinq niveaux pour éditer ou supprimer une donnée. De plus, les informations concernant les matériaux sont limitées à leur rapport hiérarchique et quelques caractéristiques ciblées (aucune information sur ses propriétés etc.).

1.3 Conclusions

Une brève présentation des interfaces a déjà permis de cerner quelques problèmes évidents de celle-ci mais elle ne saurait permettre de privilégier des pistes de solutions. A ce stade, une correction de

certaines fonctionnalités spécifiques semble suffir, mais comment en être sûr ? Quelle que soit la solution finalement adoptée, il faut une analyse plus complète de l'interface afin de comprendre, en profondeur, son fonctionnement. Cette analyse permettrait d'avoir une vision moins intuitive, et plus élaborée des problèmes de l'interface, de définir un ensemble de solutions et, enfin, de trancher en faveur de l'une ou de l'autre d'entre-elles. C'est pourquoi nous avons ensuite procédé à une rétro-ingénieurie de l'interface existante.

Chapitre 2

Rétro-ingénieurie

Contrairement au cadre d'un premier développement, l'interface qui fait l'objet de ce mémoire existe déjà. Elle existe et elle s'intègre dans une application de plus grande ampleur. Il est donc essentiel d'approcher l'application existante avec un travail de rétroingénieurie.

Comme illustré précédemment l'interface d'acquisition de connaissances est construite sur une base de données commune à l'autre interface et au moteur d'inférence. C'est donc un point central de l'application avec lequel nous travaillerons constamment. Pour cette raison, il est essentiel de comprendre le fonctionnement et l'architecture de cette base de données. La documentation qui en découlera permettra d'appréhender le développement futur de notre interface mais également tout autre développement lié à l'application, dans son ensemble, qui pourrait suivre.

Il serait malheureux de ne pas tirer des conclusions constructives de l'expérience de l'interface existante. Afin de l'améliorer lors du deuxième développement, il est important de reprendre toute l'interface et d'en faire une analyse suffisamment complète que pour mettre en évidence tout ce qui est à conserver et tout ce qui doit en être éliminé. C'est donc une deuxième étape de cette rétroingénierie que de comprendre les choix d'implémentation et de conception qui ont été faits et ce qu'ils ont apporté à l'application.

2.1 La base de données

2.1.1 Vue statique

La phase de rétro-ingénieurie prévoit par définition de partir d'informations de bas niveau, afin de remonter à des modélisations de plus haut niveau de l'existant. Pour ce faire nous sommes partis du code de la base de données et avons élaboré le schéma logique de celle-ci.

Au terme d'un affinage où les types d'associations gagnent en sémantique et où on égraine les types d'entité plus techniques (propres à l'implémentation et sans sens au niveau du domaine d'application) on obtient un schéma de plus haut niveau. Il est ensuite enrichi grâce aux connaissances du domaine d'application.

2.1.1.1 Modélisation d'une base de données

Pour les modélisations de bas niveau de la base de données le schéma logique est privilégié quel que soit l'univers de modélisation. Il permet de représenter toutes les relations structurelles de la base en faisant abstraction des spécificités de son implémentation. Afin d'illustrer notre démarche et la sémantique du schéma logique nous allons partir de l'exemple suivant. Considérons la base de données d'un dossier de patient dans un cabinet de vétérinaire. Avec une connaissance basique du contexte, et le code de la base de données, nous pouvons construire le schéma logique. Le code complet de la base de données comprend l'ensemble des structures et des informations de celle-ci. Nous nous concentrerons uniquement sur les structures. Une base de données est nécessairement constituée d'un ensemble de tables. Notre exemple, comprendra trois tables `ANIMAL, CLIENT, SOIN`). Le code déclarant la création d'une table est présenté ci-dessous. Les points de suspension cachent les propriétés du concept représenté par chaque table.

```
1 create table ANIMAL (
2       ...) ;
```

Techniquement ces propriétés se matérialisent sous la forme de champs d'informations, listés entre virgules dans la déclaration de la table :

```
create table ANIMAL (
      ID_ANIMAL numeric not null,
      Espece char(15) not null,
      Nom char(15) not null,
      Pos_Nom char(15),
      Pos_Prenom char(15),
      ...);
```

La déclaration d'un champ se fait de la manière suivante : *nom du champ*, *type de l'information que peut contenir le champ*, s'il peut être laissé vide ou non (not null). Une table comprend nécessairement un ensemble de champs identifiants qu'on appelle clé primaire. Pour les informations concernant une personne par exemple, une clé primaire pourrait être son numéro de registre national. C'est toujours une information pour laquelle on ne rencontrera jamais de doublons. Parfois le domaine d'application ne donne pas d'information de ce type-là. Dans un cabinet vétérinaire par exemple, il peut être difficile d'avoir une information identifiante pour un animal soigné. Deux animaux peuvent être de même type, de même race, de même nom, de même propriétaire, ... On peut alors les identifier par groupes de champs. On peut considérer qu'on n'aura jamais pour un même propriétaire, un animal de même nom. On désignera alors comme clé primaire le couple identifiant [nom, propriétaire]. Pour se faciliter la tâche, il arrive qu'on choisisse un identifiant dit "logique" (en général, un numéro). Ainsi, le chien Médor appartenant à Mademoiselle Wauthier sera identifié par un numéro choisi arbitrairement plutôt que par [Médor,Wauthier]. Car une fois que la clé primaire est fixée, il est impossible d'insérer une ligne dans la table qui contienne des informations de même clé primaire qu'une autre ligne préalablement enregistrée. Ainsi si un jour Mademoiselle Wauthier décidait de faire soigner son poisson rouge nommé Médor également, il serait enregistré avec un autre numéro et on éviterait de cette manière un conflit d'enregistrement. Techniquement, la déclaration d'un clé primaire s'effectue comme suit : on assigne un nom à la clé, on indique entre parenthèses les champs sur lesquels porte la clé primaire. Et le tout est ajouté sous la forme d'un ajout de contrainte : *constraint ... primary key ....* Dans notre exemple, la déclaration de la table `ANIMAL` devient par suite :

```
create table ANIMAL (
      ID_ANIMAL numeric not null,
      Espece char(15) not null,
      Nom char(15) not null,
      Pos_Nom char(15),
      Pos_Prenom char(15),
      constraint ID_ANIMAL_ID primary key (ID_ANIMAL));
```

Il existe d'autres types de contraintes que la clé primaire. Une contrainte particulièrement importante est la clé *étrangère* qui fait le lien entre différentes tables. En effet, en définitive, ce qui est intéressant, ce n'est pas seulement de présenter les caractéristiques de Médor mais également de permettre de faire le lien entre Médor et un certain nombre de soins qui lui ont été prodigués. Ainsi, pour ne pas réencoder les informations de Médor à chaque fois qu'il est traité au cabinet, on va enregistrer les soins dans une table séparée contenant un ensemble de champs identifiant l'animal auquel ils se rapportent. La clé étrangère permet d'exprimer la contrainte que les lignes dans la table soin se rapportent bien toujours à un animal dans la table animal. Techniquement, la clé étrangère peut être ajoutée dans le code qui suit la déclaration de la table avec la structure présentée ci-dessous. On «altère» une table pour ajouter une contrainte à laquelle on attribue un nom (ici REF_SOIN_ANIMAL_FK). On indique les champs sur lesquels portent les clés étrangère et finalement on indique la table qu'elle référence. Dans le cas suivant, la table `SOIN` comprend un champ `ID_ANIMAL` qui doit nécessairement correspondre à une valeur de clé primaire dans la tables `ANIMAL`[1].

```
alter table SOIN add constraint REF_SOIN_ANIMAL_FK
      foreign key (ID_ANIMAL)
      references ANIMAL;
```

[1]Notez qu'une clé étrangère référence ici une table et non un champ. Elle référence alors nécessairement la clé primaire de la table concernée.

Imaginons que nous complétions notre exemple. Le cabinet compte donc un ensemble de clients. Ils possèdent des animaux. Ceux-ci peuvent être soignés avec des soins. Chacun des concepts clés : client, animal et soin deviendra une table. Un soin pourrait être caractérisé par une nature, un traitement, un prix etc. Sous ces hypothèses, notre code complet de exemple pourrait se présenter comme dans le cadre 2.1.

```
1  create table ANIMAL (
2       ID_ANIMAL numeric not null,
3       Espece char(15) not null,
4       Nom char(15) not null,
5       Pos_Nom char(15),
6       Pos_Prenom char(15),
7       constraint ID_ANIMAL_ID primary key (ID_ANIMAL));
8
9  create table CLIENT (
10      Nom char(15) not null,
11      Prenom char(15) not null,
12      constraint ID_CLIENT_ID primary key (Nom, Prenom));
13
14 create table SOIN (
15      Nature char(50) not null,
16      Traitement char(250) not null,
17      Prix float not null,
18      ID_ANIMAL char(10) not null);
19
20 alter table ANIMAL add constraint REF_ANIMAL_CLIEN_FK
21      foreign key (Pos_Nom, Pos_Prenom)
22      references CLIENT;
23
24 alter table ANIMAL add constraint REF_ANIMAL_CLIEN_CHK
25      check((Pos_Nom is not null and Pos_Prenom is not null)
26            or (Pos_Nom is null and Pos_Prenom is null));
27
28 alter table SOIN add constraint REF_SOIN_ANIMAL_FK
29      foreign key (ID_ANIMAL)
30      references ANIMAL;
```

TAB. 2.1 – Code complet du cabinet du vétérinaire

Le schéma logique, va permettre de représenter graphiquement toutes les structures indiquées dans le code : clé étrangère, clé primaire, champs, tables sans se préoccuper de la syntaxe du langage SQL qui peut varier d'un type de base de données à l'autre. La table `animal` par exemple sera représentée comme dans la figure 2.1[2].

L'étape suivante du raisonnement vise à cheminer depuis le schéma logique, très proche de la base de données, vers les relations des différents objet qu'ils représentent dans le domaine d'application. Le domaine d'application peut, lui, être représenté au travers de différents types de modélisations présentant toutes certains avantages et inconvénients. Nous avons arrêté notre choix sur le modèle entité-association (dorénavant appelé modèle ERA). Son principal concurrent serait le diagramme de classes UML. Celui-ci comprend une modélisation plus adaptée à la modélisation de classes orientées objet (avec la définition d'opérations notamment) mais moins riche en partant d'une base de données. Il ne décrit pas le type des différentes informations que l'on y retrouve par exemple. Nous reviendrons à ce type de diagramme par la suite [8].

Le modèle ERA vise à représenter l'ensemble des objets du domaine d'application sous la forme de types d'entités, et de présenter les relations qui les lient entre elles. Ces associations sont dotées de cardinalités indiquant dans quelle proportion un type d'entité entre dans la relation. Il existe un ensemble de règles qui depuis le schéma logique permettent de construire un schéma entité association dont il pourrait être la représentation logique.

[2]L'ensemble des schémas produits dans la section vue statique ont été produits au moyen de l'outil DBMain

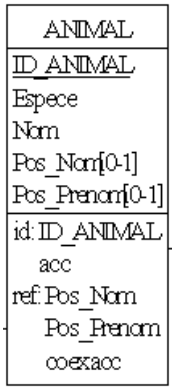


FIG. 2.1 – Représentation d'une table

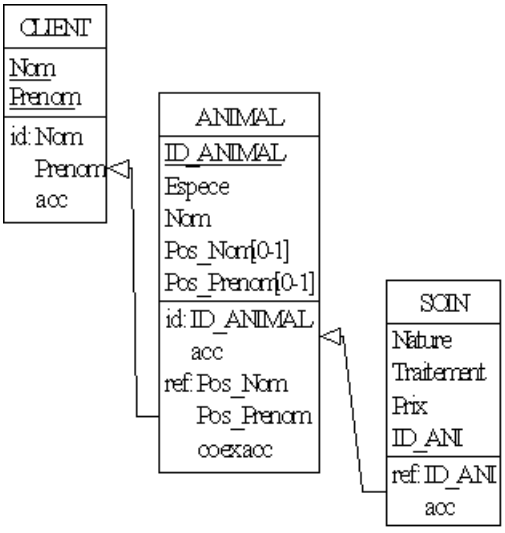


FIG. 2.2 – Exemple de schéma logique

Le schéma ERA est plus riche sémantiquement que le schéma logique. Dans le cadre d'un travail de rétro-ingénieurie cela représente une difficulté. Il faut donner du sens aux relations entre les entités qui ne sont pas directement identifiables dans le code de la base de donées. C'est là que la connaissance du domaine d'application va permettre de comprendre avec quelle logique on a tenté de représenter l'univers de l'application. Ce cheminement depuis le code vers un schéma de haut niveau d'abstraction est nécessaire pour comprendre le rapport entre la base de données et son contexte d'utilisation. C'est ce lien qui, une fois respecté, assure qu'on n'introduise pas des informations inadéquates qui auraient des répercussions sur le bon fonctionnement du reste du programme.

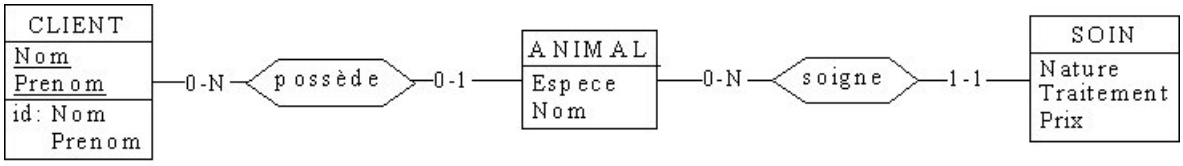


FIG. 2.3 – Exemple de schéma entité-association

La figure 2.3 exprime l'idée qu'un animal peut ou non avoir un propriétaire. Un client peut avoir un nombre entre 0 et l'infini d'animaux. Ceux-ci peuvent être soignés avec un ensemble de 0 à une infinité de soins. Remarquez que la cardinalité [0-1] associée au nom et prénom du propriétaire de l'animal dans la table `ANIMAL` a été transformée en une relation où un animal est associé à 0 ou 1 propriétaire.

2.1.1.2 Le schéma logique

En appliquant systématiquement ce raisonnement au code de la base de données d'Expesurf nous obtenons le schéma logique des figures 2.4 et 2.5. On y a transformé chaque déclaration de table et chaque déclaration de clé référentielle et de clé primaire dans son équivalent graphique. Le code de la structure de la base de données qui compte une vingtaine de pages de code indigestes peut ainsi être représenté, sans perte d'informations, en deux pages visuellement claires. Au travers de ce schéma on voit apparaître des concepts clés du domaine d'application et la position qu'ils occupent. Ainsi, la table `MATERIAU` occupe la place centrale qu'on lui reconnait également dans le domaine d'application. On remarque que la table `INCOMPATIBILITE` dépend de nombreuses autres tables tandis que certaines tables sont parfaitement isolées : `UTILISATEUR`,`TABLE_OID`... On note d'ailleurs que certaines tables ont un sens évident dans le domaine d'application tandis que d'autres relèvent de l'application informatique. Par exemple, la table `UTILISATEUR` reprend certainement les informations des utilisateurs de l'application. Elle permet d'enregistrer les coordonnées de la personne, son login... mais l'utilisateur n'intervient en rien dans le processus du domaine d'application où nous nous inquiétons de séquences de traitements à associer à une pièce. Il relève d'un choix applicatif : celui de retenir les coordonnées des utilisateurs du logiciel. S'il est très important d'avoir un schéma logique de la base de données pour visualiser l'organisation des informations, il n'est pas évident d'y repérer les structures qui dépendent de l'application de celles qui représentent des éléments clés du domaine d'application. Afin de mieux comprendre le chemin parcouru initialement depuis le domaine d'application pour arriver à ce schéma logique, il nous faut traduire le schéma actuel en schéma entité-association.

2.1.1.3 Le schéma entité-association

Une première traduction du schéma logique en schéma entité-association donne le résultat des figures 2.6 et 2.7. Nous appliquons les règles de base de traduction. Tout élément d'un schéma logique a un équivalent immédiat en schéma entité-association. Le schéma résultant n'a pas un grand intérêt par rapport au schéma logique précédent si ce n'est que le schéma entité-association permet de définir d'autres relations que la simple relation *un à plusieurs* que décrit la flèche du schéma logique. Nous allons donc pouvoir éliminer certains types d'entités au profit de relations plus complexes.

Dans ce schéma, tout un ensemble d'informations persistent qui n'ont pas véritablement de sens dans le domaine d'application. La seule simplification qui a été apportée ici est le fait de ne pas inclure les champs qui permettent normalement la confirmation par un tiers de toute modification de ce type d'entité (les champs *val_state* et *val_user*). Ils seront décrits plus en détail par la suite mais à ce stade-ci nous

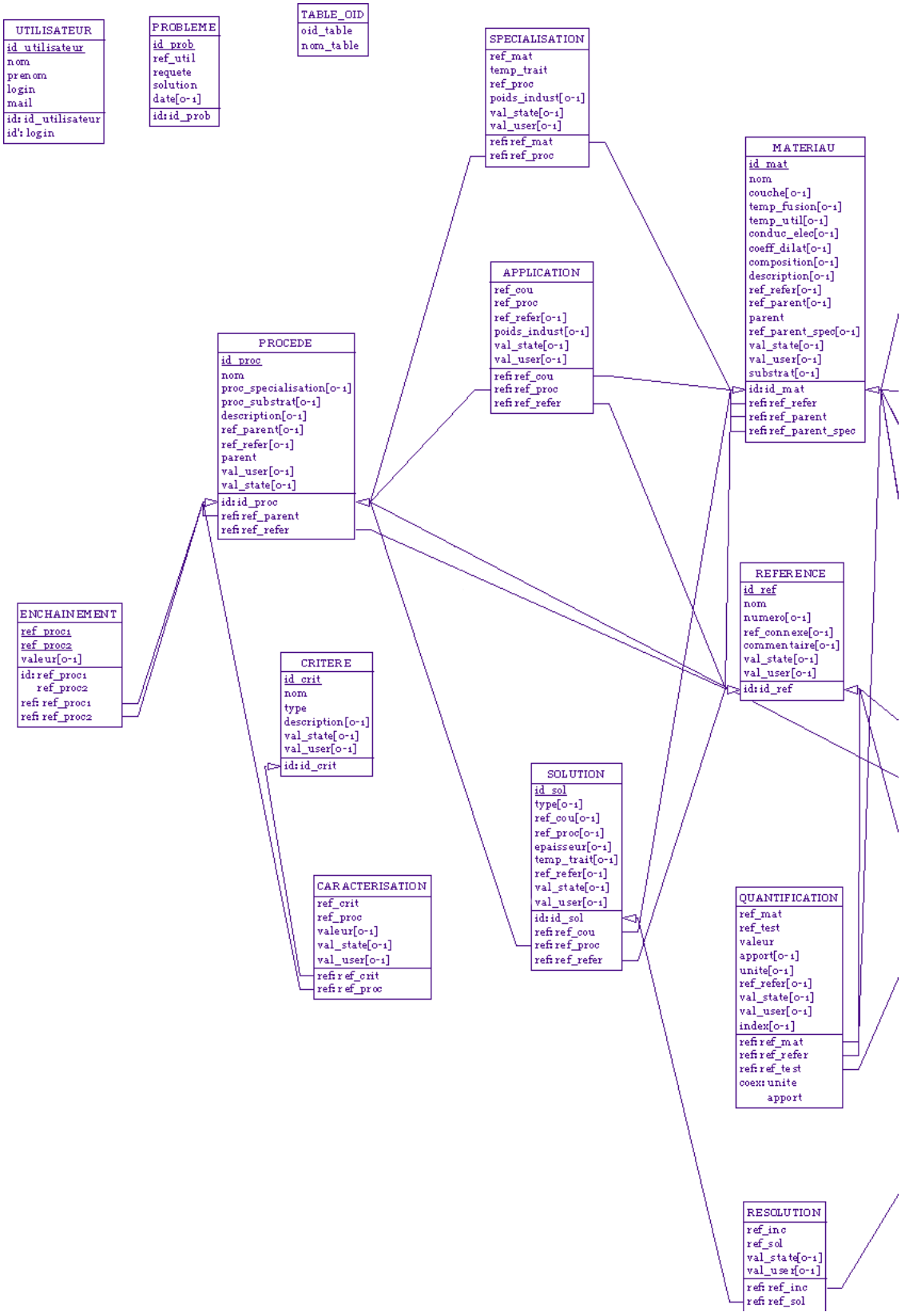


FIG. 2.4 – Schéma Logique

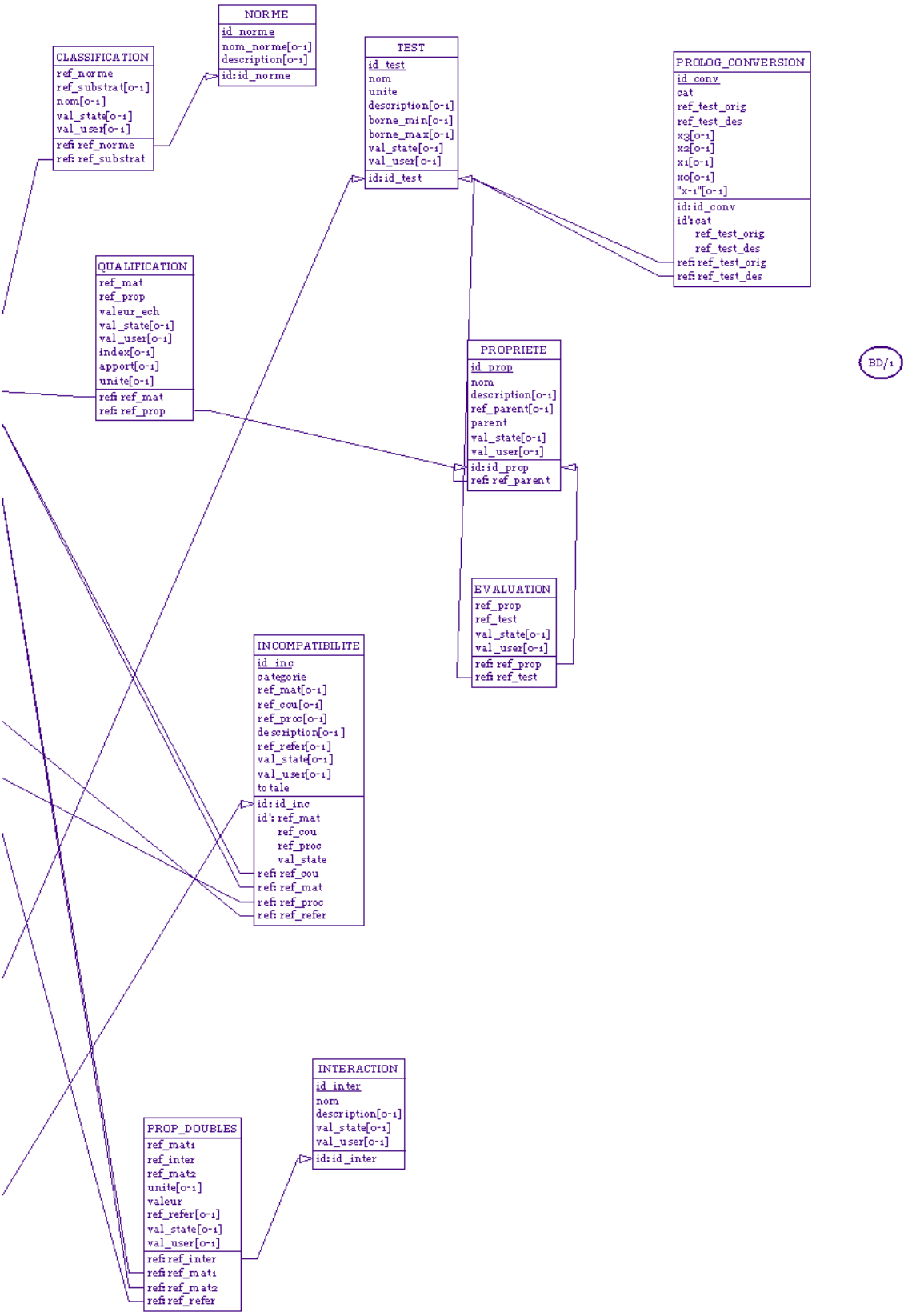


FIG. 2.5 – Schéma Logique

pouvons simplement remarquer qu'ils n'apportent pas d'information spécifique à un matériau, procédé, interaction... C'est une information purement liée à l'implémentation et qui ne fait pas partie du domaine d'application au sens stricte. L'ensemble des types d'entités faisant l'objet d'une double vérification y sont présentés en vert.

[tph]

Dans une deuxième étape, on va désolidariser le type d'entité référence qui lui aussi n'a pas d'équivalent dans le domaine d'application[3]. Dans le schéma suivant (figures 2.8 et 2.9) les types d'entités oranges et verts font l'objet d'une double vérification lorsqu'elles sont modifiées. Les types d'entités oranges peuvent, de surcroît, faire l'objet d'un référencement. Le type d'entité `APPLICATION` qui ne comportait qu'un attribut optionnel a été transformé en un type de relation caractérisé par ce même attribut. Nous pouvons nous permettre cette transformation également parce que nous reconnaissons dans le type d'entité `APPLICATION` la relation qui dit qu'un procédé s'applique à un matériau. C'est ainsi que nos connaissances du domaine d'application vont nous permettre de réduire également le type d'entité `CLASSIFICATION` en un type d'association indiquant qu'un matériau peut être «classé» selon certaines normes. Cette relation peut être qualifiée par un nom. Les types d'associations «résoud»,«caractérise»,«met en évidence» découlent toutes de ce type de raisonnement. Ces simplifications apportent un intérêt majeur : notre modélisation s'affine de plus en plus pour représenter le domaine d'application véritablement plutôt que la base de données qui en est un reflet technique.

Après quelques revérifications, nous arrivons à éliminer tous les types d'entités et associations qui sont propres à l'implémentation du système et non au domaine d'application et arrivons au schéma suivant : figures 2.10 et 2.11. Certains noms des différentes relations ont été revus pour plus de clarté. Certaines tables du schéma logique initial n'ont plus leur équivalent dans ce schéma-ci : la table de référence, d'utilisateurs, de conversions prolog... Leurs relations ont aussi disparu. Les types d'entités `PROLOG_CONVERSION` et `PROBLEME` étaient, vu le contenu de leurs tables homologues, des tables facilitant le fonctionnement du moteur d'inférence et l'enregistrement des problèmes soumis au système par les utilisateurs.

A ce stade, des simplifications peuvent encore être effectuées pour obtenir un schéma d'abstraction encore supérieur (voir figures 2.12 et 2.13). En effet, certains types d'entités, bien que qualifié par plusieurs attributs, ont un rôle associatif. Une *qualification* n'a pas de sens matériel dans le domaine d'application. Une qualification n'est que l'association d'une propriété et d'un matériau. Il en va de même pour les quantifications par exemple. De ce fait, nous pouvons ramener ces types d'entités à des types d'associations en se rapprochant encore du domaine d'application. Ce processus est essentiel dans une rétro-ingénieurie parce qu'on découvre dans le dernier schéma une représentation possible que les développeur initiaux avaient du domaine d'application. Si des éléments du domaine d'application n'y sont pas représentés c'est parce que l'on a omis de les prendre en compte dans le système expert. A l'inverse, si des incohérences ou des contradictions avec le domaine apparaissent à ce niveau on peut s'inquiéter de la bonne représentation de ces concepts dans la base de données. Au delà de l'intérêt de la démarche lors d'une rétroingénieurie, ce type de documentation servira également lors de futurs développements afin de s'assurer que les contraintes établies dans ce schéma continuent à être respectées. L'implication d'une clé étrangère dans le code d'une base de données n'est pas évident alors que dans ce schéma logique on se rend compte de l'impact qu'a la suppression d'une relation entre deux entités.

Malgré le fait que l'on peut remonter à un niveau d'abstraction supérieur on va se contenter du schéma des figures 2.10 et 2.10 pour décrire les différents types d'entités. Son niveau d'abstraction est à la fois suffisant pour identifier dans le domaine d'application la sémantique de tous les types d'entités et leurs relations, et à la fois suffisamment bas que pour que les types d'entités aient encore un équivalent évident dans le schéma logique (et donc, dans la base de données). C'est un niveau d'abstraction clé qui va permettre au développeur de se repérer plus facilement que dans un schéma ERA de très haut niveau.

Chaque type d'entité a une sémantique particuière qu'il convient de définir afin qu'aucune confusion ne soit possible quand à leur sens et utilité dans le domaine d'application. Chaque relation mérite d'être décrite avec la même rigueur. Toutes les informations qui complémentent le schéma ERA viennent directement du domaine d'application. [4]

[3]Par la suite, et malgré une bonne compréhension de l'interface existante, il fût impossible de cerner avec précision le rôle de la table `Reference` qui a une position centrale dans le schéma mais qui ne compte aucune ligne. Elle relève probablement d'une ancienne implémentation.

[4]Connaissances basées sur le rapport technique du projet [15] et sur les réunions avec le client et Monsieur Jacquet.

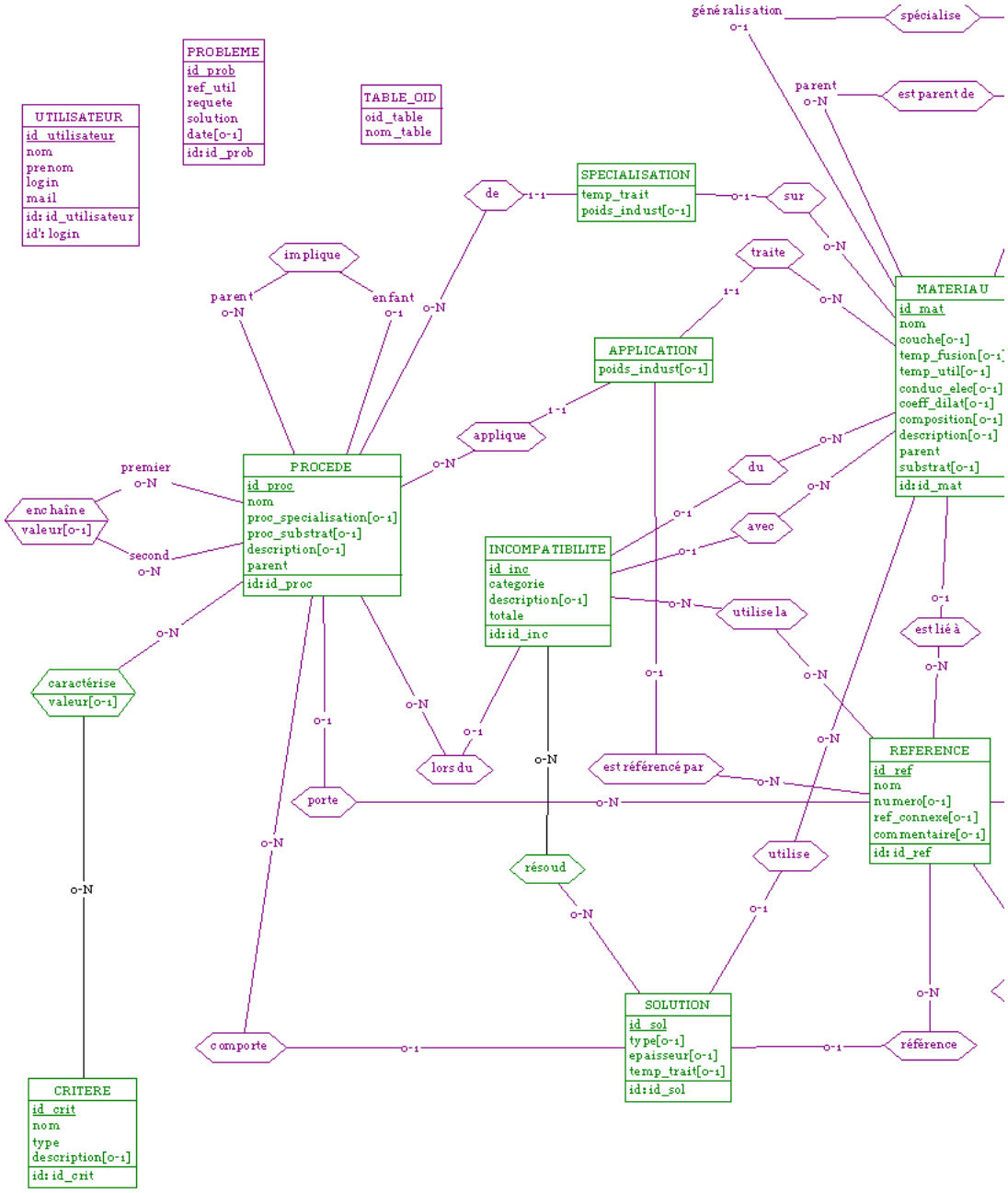


FIG. 2.6 – Schéma Entité Association 1

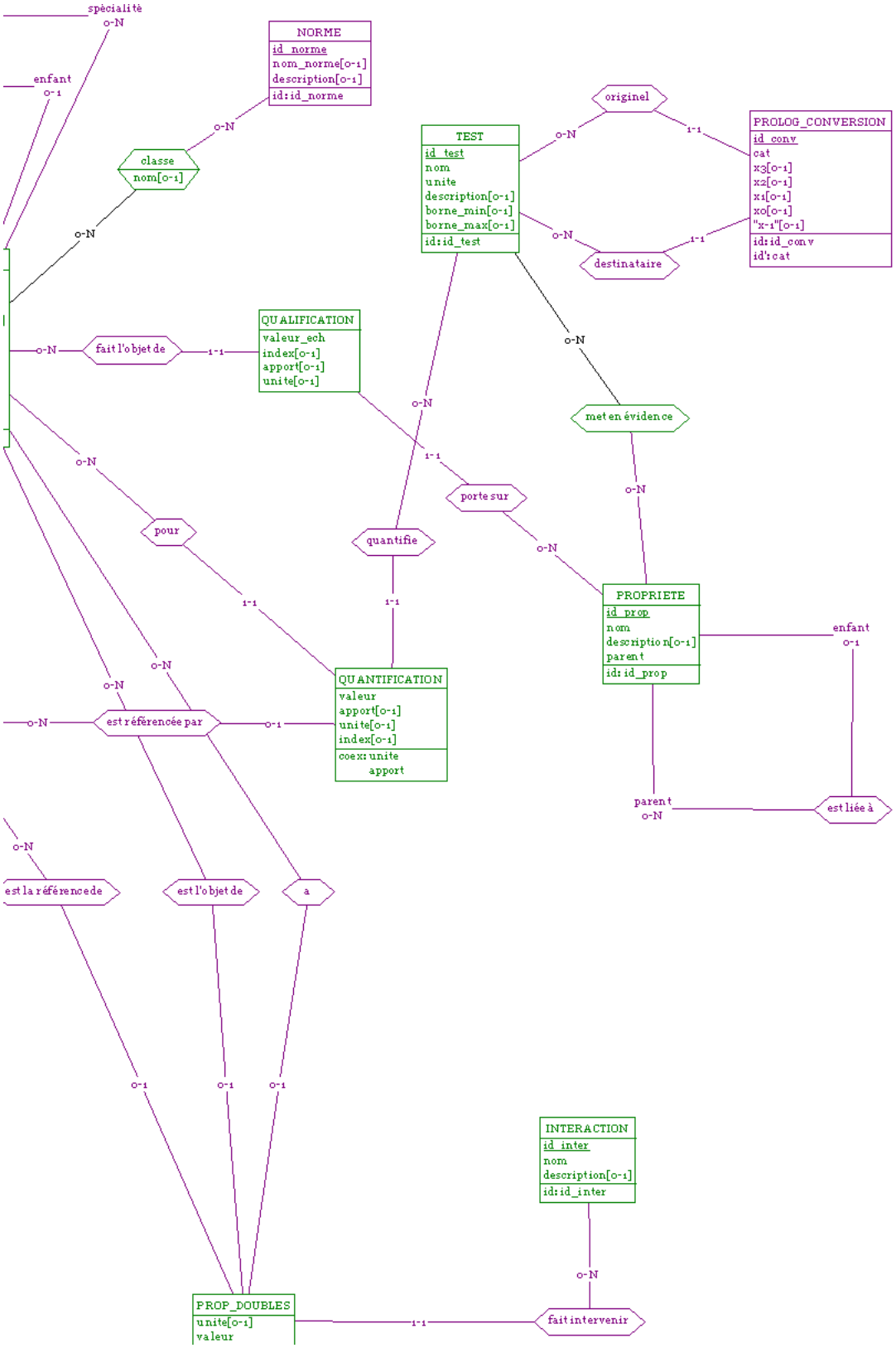


FIG. 2.7 – Schéma Entité Association 1

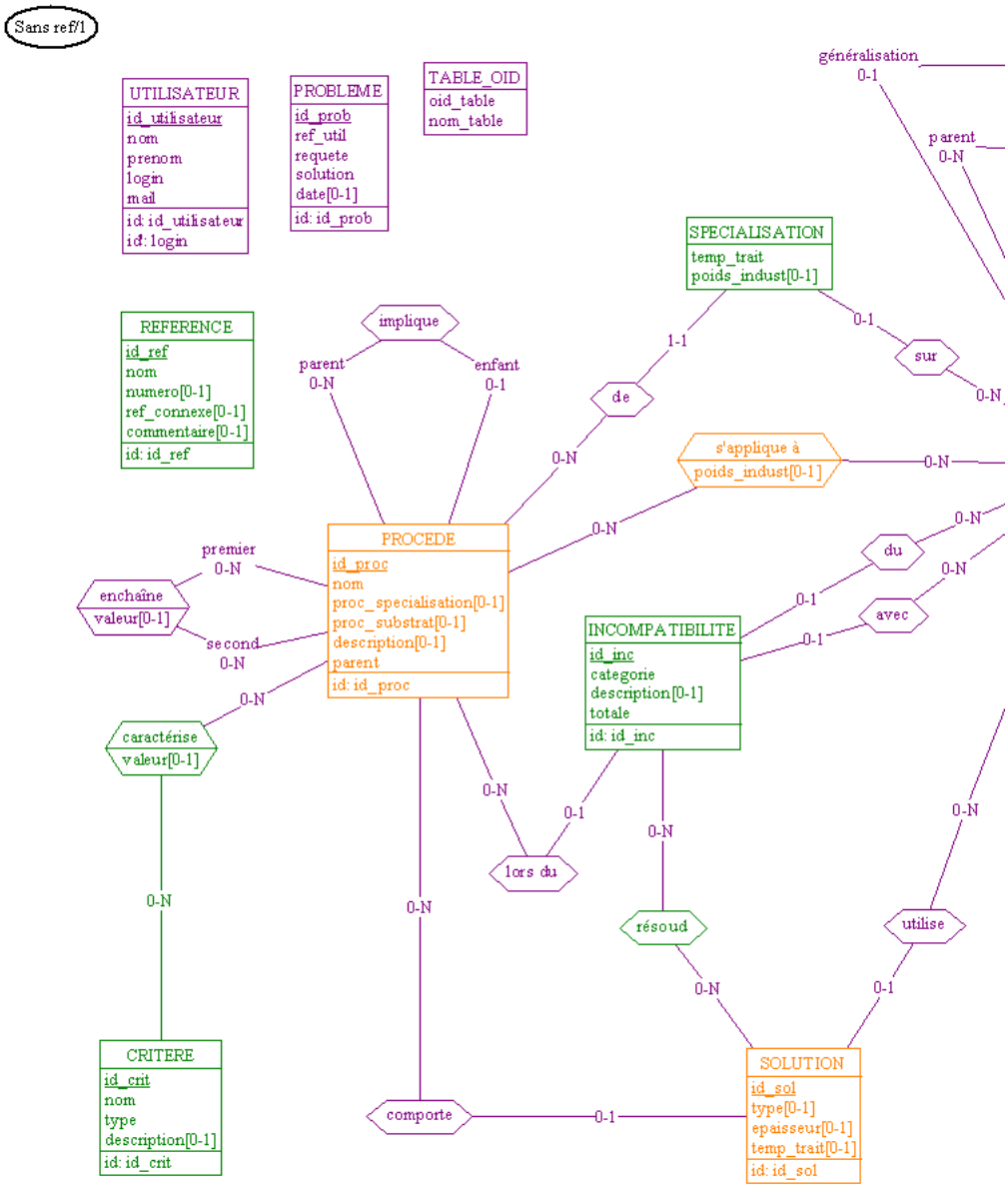


FIG. 2.8 – Schéma Entité Association 2

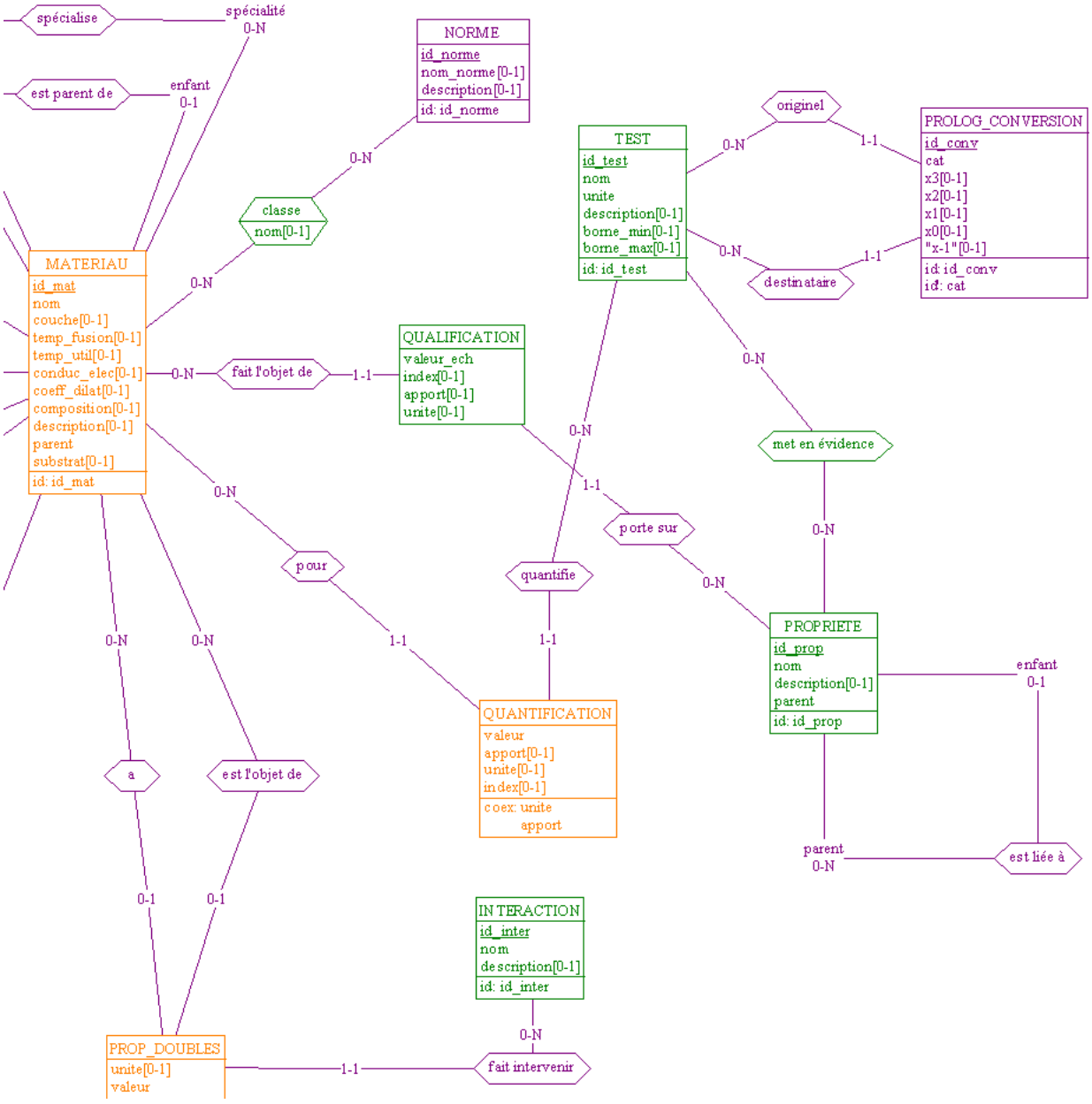


FIG. 2.9 – Schéma Entité Association 2

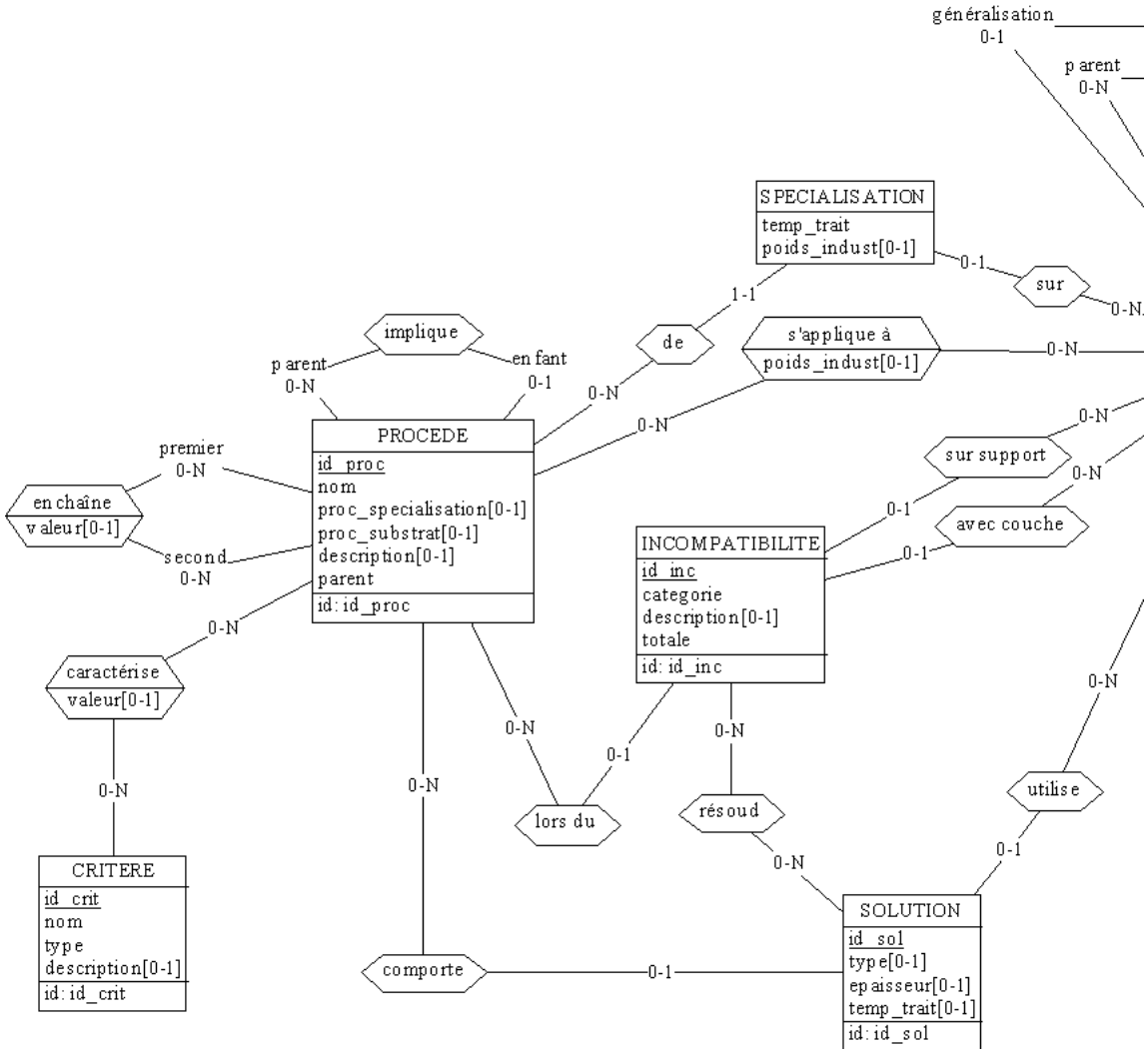


FIG. 2.10 – Schéma Entité 3

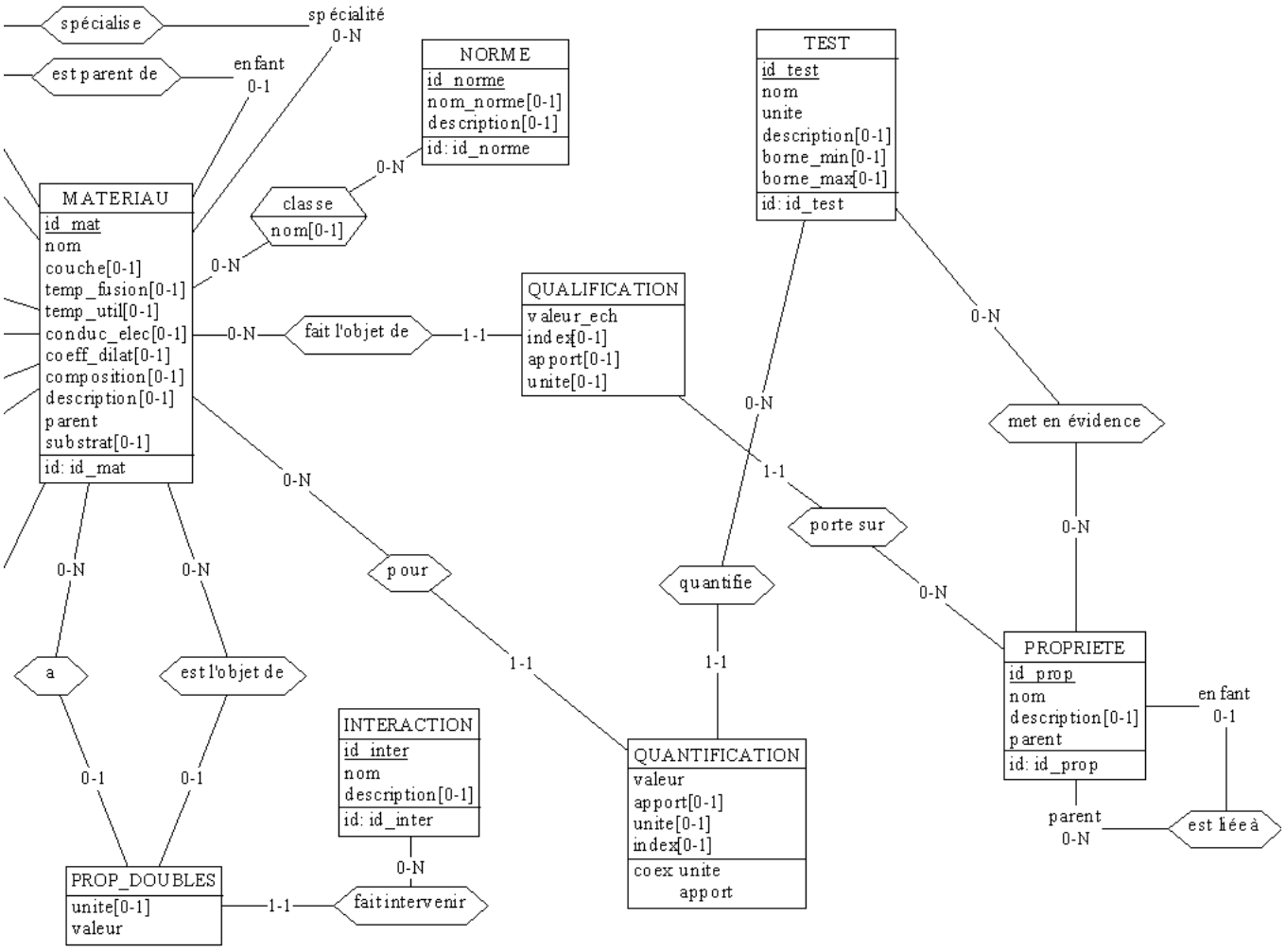


FIG. 2.11 – Schéma Entité 3

A. Description des matériaux

Les matériaux peuvent être utilisés de deux manières différentes : en tant que substrat (matériau de la pièce de base) ou en tant que couche (revêtement). Il existe un rapport familial entre des matériaux qui peuvent appartenir à une classe de matériaux qui peut elle-même appartenir à une classe de plus haut niveau encore. Les grandes familles de base comprennent par exemple les aciers, les céramiques... Toutes les familles de matériaux sont enregistrées de la même manière qu'un matériau de base puisqu'elles se comportent comme tel : dotées de certaines propriétés, interactions avec d'autres familles de matériaux... Le concept de matériau va être représenté par le type d'entité `MATERIAU`. Un matériau peut aussi être classé selon un certain nombre de normes qui lui attribuent ou non un nom particulier.

- Materiau : Il est identifié par un identifiant logique *id_mat*. Il peut être parent d'un ensemble de matériaux et enfant d'un matériau à son tour. Il peut être ou non une couche ou un substrat. Il peut spécialiser un autre matériau. Il peut faire l'objet d'une qualification qui le lie à un ensemble de propriétés. Il peut également être quantifié (Quantification) par un test. Il peut être classé selon une certaine norme qui définit d'autre manières de nommer le matériau. Outre ces relations, il est qualifié par un ensemble de champs de base : son *nom* (obligatoire), sa température de fusion (*temp_fusion*), sa température d'utilisation (*temp_util*), sa conductivité de l'électricité (*conduc_elec*), son coefficient de dilatation (*coeff_dilat*), sa *composition*, une *description*.
- Norme : Une norme classe un ensemble de matériaux. Pour une norme donnée, un matériau peut être décrit par un *nom*. La norme de son côté peut être dotée d'un nom également (*nom_norme*) et d'une *description*. Ces deux champs sont optionnels.

B. Description des propriétés

Tout un ensemble de types d'entités permet de représenter les différents types de propriétés et leurs rapports avec les matériaux. En effet, un matériau peut être doté de trois types de propriétés différentes :

- les données «booléennes» : c'est un ensemble de propriétés qu'un matériau a ou n'a pas tout simplement. Il n'existe pas de degré ou de quantification qui lie ces propriétés, elles sont juste possédées ou non.
- les données en échelle : à l'inverse des données booléennes, les données en échelle son quantifiées. Le lien entre un matériau et sa propriété est pondéré par un degré : degré de porosité, ...
- les données découlant d'une intéraction : elles reprennent l'ensemble des propriétés liées au rapport entre ce matériau et d'autres matériaux.

Les propriétés en échelle ou booléennes peuvent être mises en évidence par des tests qui peuvent faire intervenir des matériaux dans une certaine proportion.

- Prop_Doubles : Les propriétés doubles illustrent les propriétés découlant d'une interaction. Ce type d'entité quantifie la relation entre le matériau et l'interaction dont il fait l'objet au moyen d'une unité (optionnelle) et d'une valeur.
- Interaction : Une interaction décrit une interaction entre plusieurs (0-N) matériaux. Elle les fait intervenir dans certaines proportions décrites par le type d'entité `PROP_DOUBLES`. Deux champs caractérisent une interaction : *nom*, *description* (optionnel).
- Qualification : Le type d'entité qualification décrit la relation entre un matériau et une propriété. C'est une relation dotée de certains paramètres tels que la valeur d'échelle (s'il y en a une : données d'échelle ou booléenne), un index, un apport et une unité.
- Propriete : Une propriété peut représenter une propriété booléenne ou en échelle. C'est son lien avec la table qualification qui va permettre de définir ce rapport. Elle peut être mise en évidence par un test. Il existe pour les propriétés comme pour les matériaux un classement hiérarchique. Une propriété peut être un sous-type d'une autre propriété. Elle hérite alors de ses caractéristiques. Elle est décrite par son *nom*, une *description* (optionnelle) et une propriété *parent* (indique si la propriété est parente ou non).
- Quantification : La quantification associe à un test l'ensemble des matériaux qu'il fait intervenir. Il décrit les proportions dans lesquels ces matériaux interviennent au moyen de 4 champs : *valeur* (champ obligatoire), *apport*, *unite*, *index*.
- Test : Le type d'entité test représente un test comportant un *nom*, une *unite*, une *description*, une borne minimale(champ *borne_min*) et une borne maximale (champ *borne_max*). Ces trois derniers

champs sont optionnels.

C. Description des procédés

Une bonne partie de la singularité d'Expesurf se situe au niveau du traitement des procédés. En effet, il est relativement simple de prévoir pour un matériau donné l'ensemble des procédés qui le doteraient d'une propriété donnée. En revanche, le fait de travailler avec des ensembles de couches successives impose de distinguer les couches appliquables à un substrat et celles appliquables à une autre couche. De ce fait à l'instar de ce qu'on a rencontré dans la table matériau, un procédé s'appliquer à un substrat ou à une couche. De plus, tout comme les propriétés et les matériaux, on peut classer les procédés de façon hiérarchique en partant de procédés de plus haut niveau vers le plus bas niveau. Ici encore, si conceptuellement on serait tenté de distinguer des classes de procédés de base, la représentation ne serait pas adéquate puisque les catégories de procédés peuvent avoir exactement les mêmes propriétés et associations qu'un procédé de base.

- PROCEDE : Ce type d'entité reprend l'ensemble des procédés qui peuvent être des substrats ou non (*proc_ substrat*) ou un procédé de spécialisation (*proc_ specialisation*). Il est nécessairement caractérisé par une description et s'applique à un matériau (dans le cas d'un substrat) ou spécialise un matériau dans le cas d'une spécialisation. Un procédé peut consister en un enchaînement de procédés. Cette association est pondérée par une *valeur* (optionnelle).
- CRITERE : Un procédé peut également être caractérisé par un certain nombre de `CRITERE`. Ce type d'entité comprend un *nom*, une *description* (optionnelle) et un *type*.
- SPECIALISATION : Un procédé peut spécialiser un matériau. Cette spécialisation est caractérisée par une température de traitement (*temp_ trait*) et un poids industriel (*poids_ indus*, optionnel).

D. Description des incompatibilités

Les matériaux et les procédés peuvent faire l'objet d'incompatibilités. Ces incompatibilités excluent de l'ensemble des solutions certaines combinaisons de procédés, de couches, et de substrats. Certaines peuvent être solutionnées au travers de l'application d'un procédé ou d'un matériau.

- INCOMPATIBILITE : Trois paramètres peuvent être la source d'une incompatibilité : le substrat, la couche ou le procédé. Une incompatibilité est dite *totale* lorsqu'un des paramètres (substrat, couche ou procédé) est absent. Si l'incompatibilité est totale elle intervient alors pour toute valeur du champ non défini (pour toute propriété si la propriété est absente, ...). Une incompatibilité peut intervenir de différentes manières : pour un substrat donné avec un procédé, pour une couche donnée avec un procédé donné, entre deux matériaux, ou encore pour un substrat, un procédé et une couche donnée. Dans les trois premiers cas de figure, l'incompatibilité peut être définie comme *totale* ou non. Cette incompatibilité peut être ou non résolue par un ensemble de solutions. L'incompatibilité est décrite par sa *categorie*, une *description* (optionnelle) et le fait qu'elle est *totale* ou non.
- SOLUTION : Une solution relie un procédé et/ou un matériau à un ensemble d'incompatibilités. L'application de la solution résoud les incompatibilités auxquelles elle est liée. Elle possède quelques caractéristiques propres telles que son *type*, son *epaisseur*, et sa température de traitement (*temp_ trait*). Ces trois attributs sont facultatifs.

Finalement, si cette représentation rend bien compte du rapport entre les éléments présentés par la base de données et le domaine d'application, il est possible de remonter un niveau supplémentaire dans les degrés d'abstraction pour obtenir un schéma ERA qui correspond véritablement à une modélisation du domaine d'application et qui s'abstrait complètement de l'implémentation qui en a été faite et dont nous sommes partis. Un ensemble de types d'entités possèdent encore un identifiant logique : *id_ mat*, *id_ sol*... Ces identifiants ne représentent rien du point de vue du domaine d'application. De plus, certains types d'entités n'ont d'intérêt que le fait qu'ils lient deux autres types d'entités (`PROP_DOUBLES`, `SPECIALISATION`, `QUANTIFICATION`, ...). On sent bien que ces types d'entités ont une valeur bien moins importantes que `MATERIAU` ou `PROPRIETE` par exemple et qu'ils mériteraient d'être réduits à un type d'association muni de quelques attributs. Ces dernières modifications mènent au schéma 2.12 qui n'apporte que peu informations directement liées à la base de données dont on est partis et, à l'inverse, modélise le domaine d'application au complet.

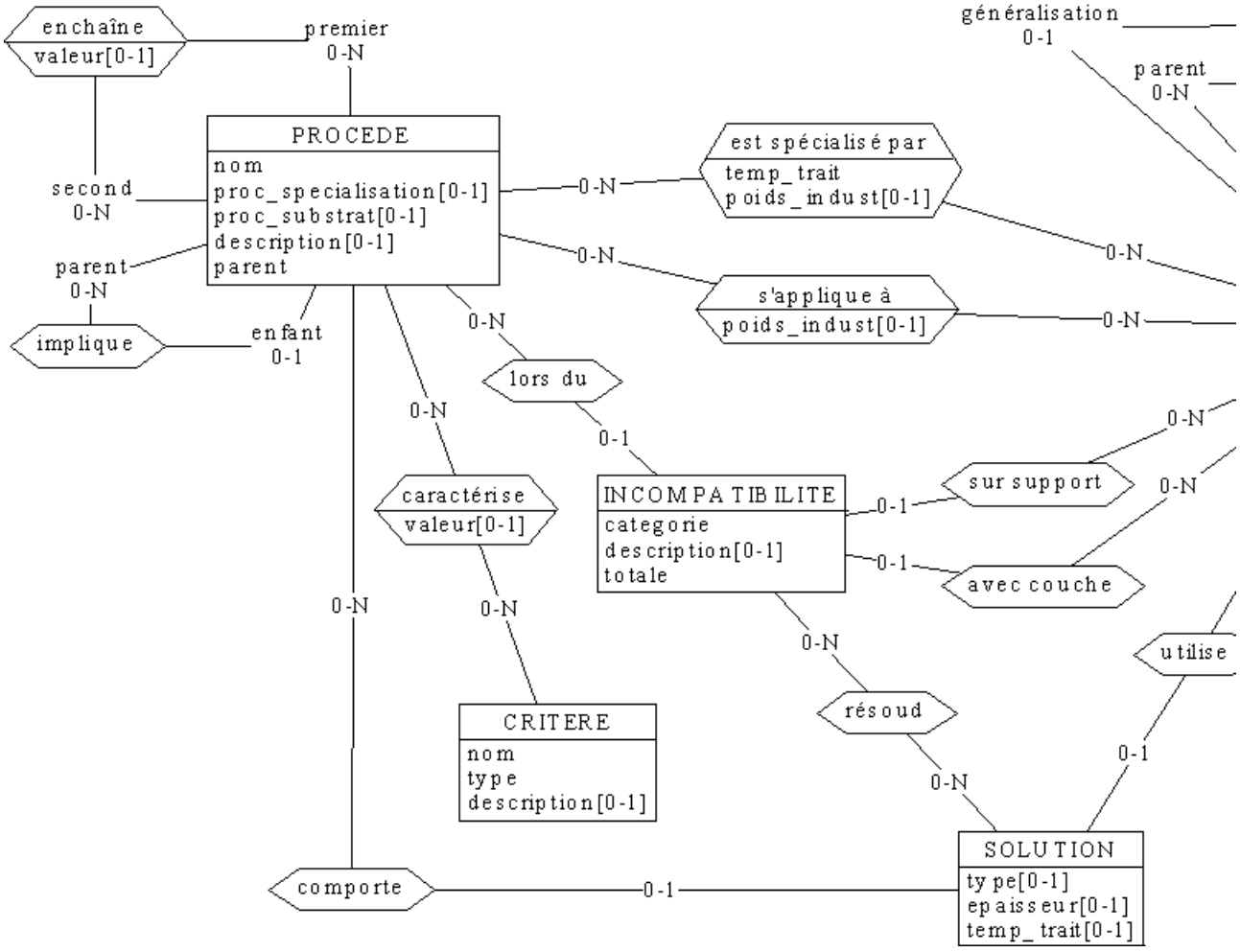


FIG. 2.12 – Schéma Entité Association Final

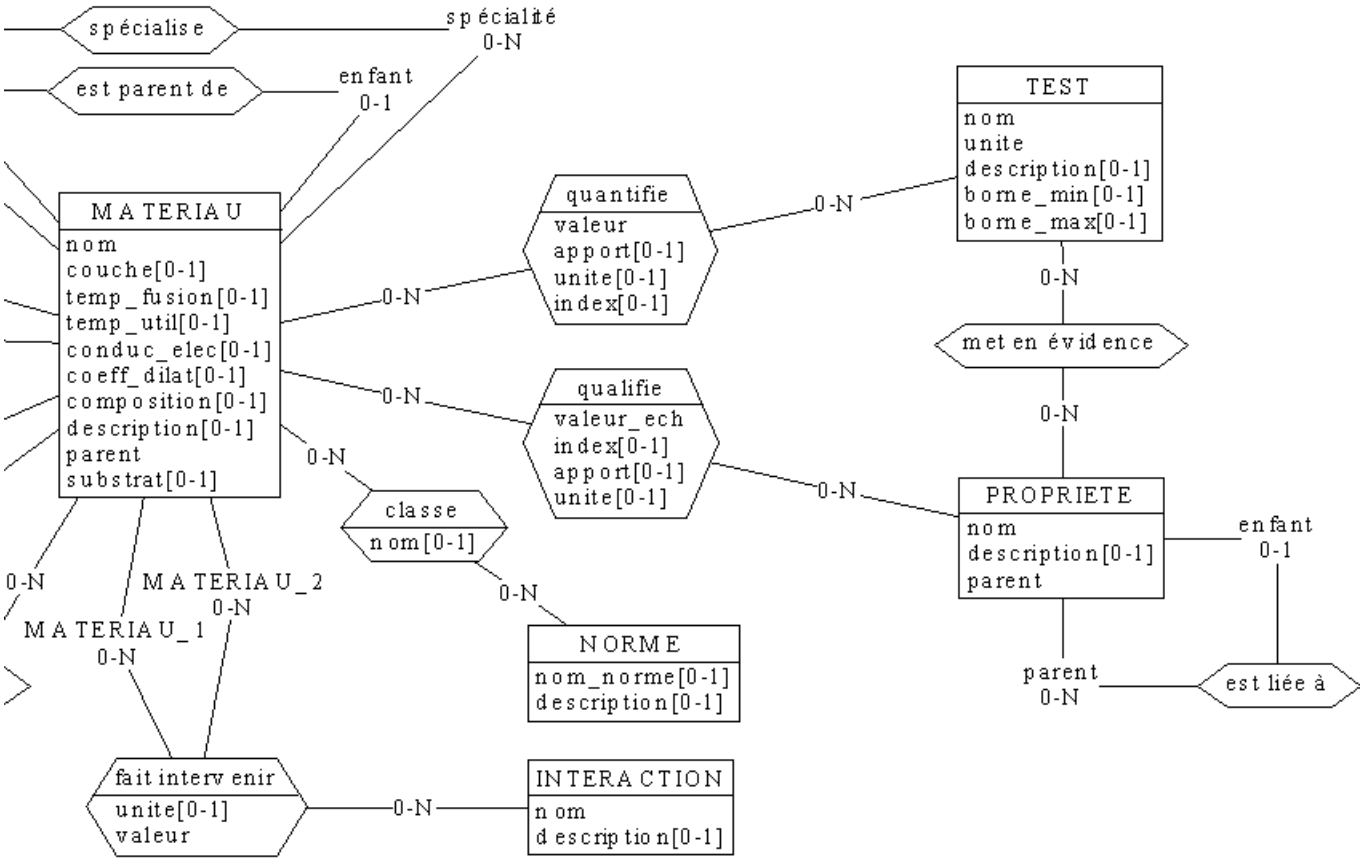


FIG. 2.13 – Schéma Entité Association Final

2.1.2 Vue dynamique

2.1.2.1 Les triggers et fonctions

Il serait insuffisant de se contenter d'une vision structurelle de la base de données en place. En effet, pour garantir l'intégrité de la base de données des parts inégales de traitements et de vérifications sont réparties entre

- l'interface à l'encodage (dont le code permet de vérifier certaines contraintes et de délimiter l'ensemble des opérations de l'utilisateurs),
- au travers de la structure de la base de données (types des champs, clés de référence, clés uniques etc.)
- et enfin au travers d'un ensemble de *triggers* .

La base de données ne comprend pas que des données et des définitions de structures comme on a pu le voir auparavant. Elle comprend également un ensemble d'éléments procéduraux qui vont effectuer un ensemble d'opérations lors de leur appel (dans le cas d'une fonction) ou d'un évènement clé (cas des triggers qui est une fonction qui s'exécute automatiquement dans un contexte donné). Les triggers sont déclarés de la manière suivante dans le code de la base de données :

```
1 create trigger monTrigger before update
2 on maTable for each row
3 execute procedure maFonction()
```

On crée ici un trigger (*create trigger*) de nom «monTrigger» qui va se déclencher, avant toute mise à jour (*before update*) de la table «maTable» et pour chaque ligne concernée par l'opération (*for each row*), la fonction «maFonction». La fonction, elle, peut être écrite dans différents langages de programmation.

Dans la mesure où nous souhaitons supplanter l'interface existante il est important d'identifier la frontière spécifique entre les vérifications apportées par la base de données, et celles apportées par l'interface. Nous allons pour ce faire décrire précisément les actions des différents triggers et fonctions ainsi que le rapport entre cet impact et les contraintes d'intégrité correspondantes dans la base de données.

La base de données comprend deux différents types de triggers. Certains triggers assurent la vérification de contraintes d'intégrité liées au domaine d'application :

- Trigger : `app_cou`, Fonction : `sol_cou`
  description : vérifie que la couche référencée dans l'application/la solution que l'on s'apprête à insérer/mettre à jour correspond bien à l'enregistrement d'une ligne dans la table *materiau* définie comme une couche), renvoie une exception sinon.
  déclenchement : avant toute modification ou insertion, pour chaque ligne concernée
  table concernée : *application*, *solution* (respectivement)

- Trigger : `app_proc`
  description : vérifie que le procéde de l'application que l'on s'apprête à insérer/mettre à jour correspond bien à l'enregistrement d'une ligne dans la table *procede* spécifiée en tant que procédé d'application et non de spécialisation, renvoi d'une exception sinon
  déclenchement : avant toute modification ou insertion, pour chaque ligne concernée
  table concernée : *application*

- Fonction : `spec_proc`
  description : vérifie que le procéde de la spécialisation qu'on s'apprête à insérer/mettre à jour correspond bien à l'enregistrement d'une ligne dans la table *procede* spécifiée en tant que procédé de spécialisation et non d'application, renvoi d'une exception sinon
  déclenchement : avant toute modification ou insertion, pour chaque ligne concernée
  table concernée : *specialisation*

Les fonctions `spec_proc` et `sol_cou` ne sont pas utilisées pour l'instant. Elles prendront davantage de sens à l'avenir si la hiérarchie familiale de la base de données est abandonnée. Toutefois, actuellement, les solutions et les spécialisations pouvant faire intervenir des matériaux parents (ni couche, ni substrat) elles ne sont pas possibles.

Pour une meilleure sécurité, un système de contre-validation des insertions/modifications/suppressions a été mis en place. Il est entièrement géré au travers de trois triggers et six fonctions. Les tables concernées par les fonctions sont celles sur lesquelles elles agissent lorsqu'elles sont appellées. L'appel sur une autre table n'aurait pas d'impact sur ses enregistrements.

- TRIGGER : `stateOnDelete`, `stateOnInsert`, `stateOnUpdate`
  DESCRIPTION : réduit une insertion/suppression/modification à une demande d'insertion/suppression/modification requérant validation selon le diagramme d'états en figure 2.14.
  DÉCLENCHEMENT : avant une suppression, une insertion, ou une modification respectivement et pour chaque ligne concernée
  TABLES CONCERNÉES : *propriete*, *test*, *critere*, *application*, *evaluation*, *specialisation*, *qualification*, *materiau*, *procede*, *quantification*, *prop_ doubles*, *incompatibilite*, *solution*, *caracterisation*, *interaction*, *reference*, *resolution*, *classification*

- FONCTION : `accept_insert`, `accept_update`, `accept_delete`
  PARAMÈTRES VARYING_CHAR,VARYING_CHAR : soit respectivement le nom de la table où dont on veut valider l'enregistrement, l'identifiant de l'enregistrement concerné (un nombre). La fonction ne prenant qu'un identifiant numérique unique, pour la table *prop_ doubles* sur laquelle agit la fonction, la référence au premier matériau est séparée de la référence au second matériau par ' :' [5]
  DESCRIPTION : valide l'action des opérations d'insertion, mise à jour, et suppression respectivement. La validation d'une mise à jour remplace l'enregistrement existant par l'enregistrement de mise à jour. La suppression, agit en cascade. La validation de l'ajout d'un enregistrement valide également l'ajout des enregistrements qui en dépendent (le fait de valider l'enregistrement d'un matériau entraine la validation de l'enregistrement de ses qualifications, quantifications et classifications).
  DÉCLENCHEMENT : à l'appel [6]
  TABLES CONCERNÉES[7] : *propriete*, *test*, *critere specialisation*, *materiau*, *procede*, *prop_ doubles*, *incompatibilite*, *solution*, *interaction*, *resolution*, *classification*

- FONCTION : `cancel_insert`, `cancel_update`, `cancel_delete`
  PARAMÈTRES VARYING_CHAR,VARYING_CHAR : soit respectivement le nom de la table où dont on veut valider l'enregistrement, l'identifiant de l'enregistrement concerné (un nombre). La fonction ne prenant qu'un identifiant numérique unique, pour la table *prop_ doubles* sur laquelle agit la fonction, la référence au premier matériau est séparée de la référence au second matériau par ' :'.
  DESCRIPTION : annule l'action des opérations d'insertion, mise à jour et suppression respectivement. L'état de l'enregistrement est rétabli à l'état valide (0). L'annulation d'une mise à jour supprime l'enregistrement avec les informations de mise à jour. L'annulation de l'ajout d'un enregistrement annule également l'ajout des enregistrements qui en dépendent.
  DÉCLENCHEMENT : à l'appel
  TABLES CONCERNÉES : *propriete*, *test*, *critere specialisation*, *materiau*, *procede*, *prop_ doubles*, *incompatibilite*, *solution*, *interaction*, *resolution*, *classification*

La validité des transitions entre les états (voir figure 2.14[8]) est assurée par les triggers `stateOnDelete`, `stateOnInsert`, et `stateOnUpdate` et les six fonctions décrites ci-dessus qui peuvent confirmer ou infirmer la modification demandée. L'enregistrement existe dès sa demande d'insertion jusqu'à la validation de sa suppression. Lors d'une demande d'ajout, de mise à jour ou de suppression, le trigger adapté à l'opération est actionné. Il met l'enregistrement concerné dans un état intermédiaire en attente de validation [9]. La nature de l'état dans lequel se trouve l'enregistrement à un instant donné est décrit par

[5] Un appel de confirmation d'insertion sur la table prop_doubles pourrait être : *accept_ insert("prop_ doubles","12 :15")*. Dans ce cas l'insertion préalablement demandée d'une propriété double entre le matériau 12 et le matériau 15 (*ref_ mat1* et *ref_ mat2* respectivement et non l'inverse) référencés dans la table matériau sera rendue définitive.

[6] La fonction doit être appellée pour rendre définitive (statut mis à 0, ou supprimé) une demande d'opération préalablement insérée.

[7] La validation de la suppression a visiblement un jour agi sur une table *substrat* qui n'existe plus dans la base de données mais dont la confirmation de suppression est toujours prévue dans la fonction.

[8] Le diagramme ne compte que les enregistrements effectifs et non les enregistrements comptant les informations de mise à jour.

[9] L'état 1 correspond à une demande d'insertion, l'état 2 à une demande de mise à jour, et l'état 3 à une demande de suppression.

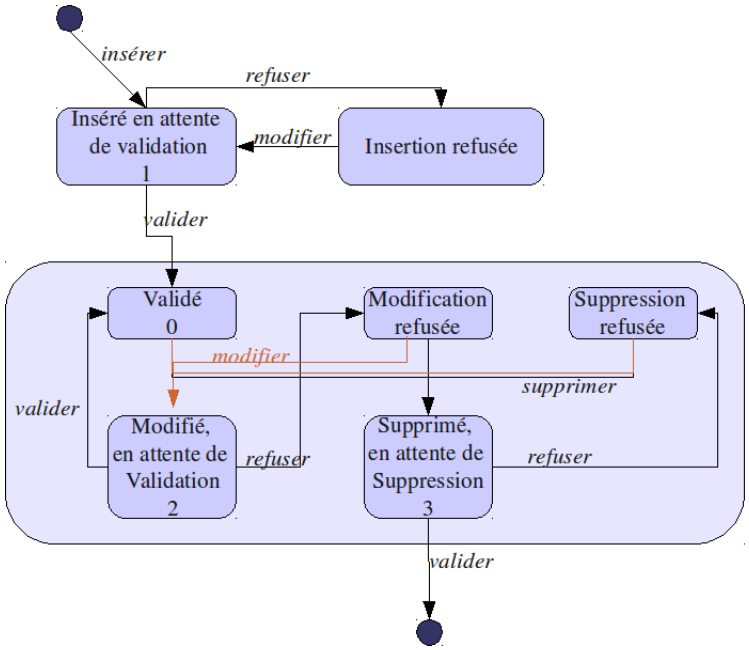


FIG. 2.14 – Diagramme d'état des enregistrements, explications à la page 37

le champ `val_state` où l'on insère le numéro correspondant à son état courant. Un état supplémentaire est utilisé : 4. Il s'agit d'un état "update" qui identifie un enregistrement contenant les modifications à apporter à un enregistrement donné. Son identifiant est l'opposé mathématique de l'identifiant de la ligne dont il contient les modifications. Ainsi si l'on veut modifier l'enregistrement du matériau 135, les données du nouvel enregistrement sont enregistrées dans un enregistrement indépendant, dans la même table, sous l'identifiant -135 qui disparaîtra lorsque la modification sera annulée ou confirmée. Ce choix implique également une impossibilité : celle de réclamer deux modifications d'un enregistrement ou de vouloir modifier une demande de changement par exemple.

Il est intéressant de remarquer que l'ensemble des tables pouvant faire l'objet d'une validation ou annulation n'est pas le même que celui sur lequel peut porter une demande de modification, d'ajout ou de suppression. En effet, les fonctions de validation sont fort proches de l'interface existante. Elles distinguent un groupe de tables «dominantes» (celles sur lesquelles agissent les fonctions de validation) des autres tables. Toutes les autres tables dépendent d'une table dominante et toute demande de modification sur celles-ci ne sera validée que lorsqu'une demande de modification d'une table parente sera validée. Par exemple, la validation d'une résolution n'est effective que lors de la validation de l'incompatibilité dont elle dépend. Ainsi, toute modification des tables dépendantes implique l'insertion d'une modification chez la table dominante dont elle dépend. Lorsque l'enregistrement parent sera validé alors la modification chez ses tables dépendantes seront validées également. Cela apporte une limitation supplémentaire : admettons que nous réclamions la modification de la qualification d'un matériau, nous ne pouvons pas ensuite vouloir modifier une autre caractéristique du matériau telle que sa description ou une quantification tant que la modification de la qualification n'a pas été validée. Enfin, si la validation fonctionne en cascade, les triggers gérant les états n'en font pas autant. Ainsi, il revient au programmeur de l'interface de créer les demandes de modifications/suppressions/mises à jour dans les tables parentes tandis que la base de données se charge de répercuter une validation sur les enregistrements dont il dépend.

Un dernier effet important de cet ensemble de fonctions est qu'il doit être contourné lors d'opérations automatiques. Lors de la validation d'une suppression, la suppression des enregistrements qui la référencent s'exécute en cascade (par exemple : la suppression d'un matériau entraîne la suppression des qualifications, quantifications, ou classifications dont il fait l'objet). La validation d'une insertion valide également d'autres insertions en cascade. Cette exécution serait empêchée par les triggers sur les autres tables. Une fonction supplémentaire qui définit les dépendances entre les tables et désactive les triggers

problématiques est appellée à chaque exécution d'une validation pour contourner les triggers gérant les états et les rétablir après l'exécution.

Notons enfin que les triggers assurent la double vérification de toutes les opérations mais ne réclament pas que la personne validant l'opération soit différente de la personne l'ayant réclamée. C'est une fonctionnalité qui était attendue de l'interface mais qu'elle ne prévoit pas.

2.1.2.2 Vues

Les vues sont en fait une un sous-ensemble des informations de la base de données. On les construit comme une requête à la base de données et on demande que le résultat de cette requête soit à tout moment disponible. Ces résultats font parfois l'objet de certaines optimisations dépendant du gestionnaire de bases de données afin qu'on puisse y accéder plus vite. La vue pourra être consultée comme si elle constituait une table à part entière.

Si les vues ne présentent pas d'intérêt structurel, dans le cadre du développement d'une interface elles font partie d'outils essentiels. En effet, non seulement le choix des vues nous éclaire sur ce que le programmeur précédent à jugé intéressant d'isoler comme type d'information mais en plus, il nous sera possible de réutiliser ces vues par la suite. De ce fait une brêve description de l'information qu'elles présentent peut être intéressante pour le travail de développement qui suivra. Nous avons identifié six types de vues différentes.

2.1.2.3 Vues de complétion

Elle fait une jointure entre la table de même nom que la vue et toutes les tables qu'elle référence. Elle présente, par exemple, non pas le contenu de la table incompatibilité seul mais également les descriptions des procédés et matériaux concernés par chaque ligne de la table.

VUE	TABLES CONCERNÉES
`current_incompatibilite`	*incompatibilite, materiau, procede*
`current_materiau`	*materiau, specialisation*

A. Vues des enregistrements réels

Les tables concernées par ces vues font toutes l'objet d'une double vérification pour toute suppression, insertion, ou modification. Elles contiennent donc deux types d'enregistrements : les enregistrements réels, et les enregistrements contenant les informations de mise à jour de la table. Ces vues reprennenent l'information des enregistrements dits "réels" de la table.

```
get_application
get_caracterisation
get_classification
get_evaluation
get_prop_doubles
get_qualification
get_quantification
get_resolution
get_specialisation
```

B. Vues informations de mises à jour

Ces vues reprennent l'ensemble des enregistrements de la table de même nom que la vue complétées des informations des tables qu'elle référence, et qui constituent des mises à jour d'informations qui n'ont pas encore été faites. A noter que l'on inclut dans cette vue les informations de mise à jour de toutes les tables concernées.

VUE	TABLES CONCERNÉES
`upd_incompatibilite`	*incompatibilite, procede, materiau*
`upd_materiau`	*materiau, specialisation*

C. Vues des demandes de mise à jour

Ces vues sont un peu le complément des précédentes. Elles couplent les informations des lignes d'information avec les informations de mise à jour dont elles font l'objet. Le statut de la ligne y est écrit en toutes lettres : "updated, inserted, deleted...".

```
view_critere
view_incompatibilite
view_interaction
view_materiau
view_procede
view_propriete
view_solution
view_test
```

D. Vues de mise à jour du moteur d'inférence

Bien qu'il semblait important de mentionner leur existance, les deux derniers types de vues ne concernent pas notre partie du système expert Expersurf. En effet elles facilitent le travail du moteur d'inférence et peut-être de l'interface d'acquisition de données qui en dépend. Nous ne détaillerons pas ce qu'elles représentent puisque ce n'est pas évident à cerner sans se plonger dans ces autres composants de l'application. Cependant si cette rétro-ingénieurie venait à servir à d'autres fins, le lecteur pourra s'intéresser à ces dernières vues.

2.2 L'interface d'acquisition de connaissances

Dans un deuxième temps, il était important de comprendre l'interface d'acquisition de connaissances en tant que telle. La documentation existante fait état de l'utilisation du framework Zope et décrit l'ensemble des structures développées en son sein pour les besoins de l'application. Néanmoins, une approche plus approfondie est nécessaire pour pouvoir appréhender la maintenance du logiciel. Il nous faudra donc mettre en évidence la structure logique de l'application et l'organisation des tâches dont elle fait l'objet afin d'en isoler les avantages et travers ainsi qu'afin de veiller à garder ou pallier ceux-ci respectivement dans la deuxième phase du mémoire.

2.2.1 Présentation de Zope

Zope est un framework qui permet le développement d'applications Web dont le succès est en grande partie basé sur le système de gestionnaire de contenu Plone qui l'utilise et qui est très répendu. A l'origine un logiciel propriétaire il est diffusé sous la licence ZPL (une licence reconnue par GPL et répondant aux normes open source). Sa première version sort en 1998 sous le nom Zope et est une fusion de trois logiciels initiaux. Le premier proposait une publication assistée sur le net, l'autre un système de bases de données orientées objet, et la troisième application des templates de texte. Les trois applications étaient développées en Python (à l'exception de certains fragments écrits en C pour plus d'efficacité à l'exécution). Depuis lors cette version a été supplantée par Zope 3 et bien que la première reste la plus répendue, elle n'est plus maintenue. L'interface d'acquisition de connaissances d'Expesurf utilise la première version de Zope d'où l'intérêt de bien comprendre son fonctionnement singulier.

2.2.1.1 Architecture et philosophie de Zope

Zope est un framework. A ce titre il propose une "fondation" sur laquelle on peut implanter son application web. Dans cette fondation il prend en charge un ensemble d'opérations clé de l'application : contrôle d'accès, la persistence des données, et également le texte XML et HTML : par le biais de page templates (ZPT).

Zope est également développé de façon orienté objet. La philosophie qui guide le raisonnement orienté objet du framework part de la constatation suivante : une URL est en fait juste une adresse référençant un objet contenu dans un ensemble de contenus imbriqués. L'accès à une URL revient à solliciter un objet pour en obtenir une réponse, une information : en l'occurrence, une page HTML [9]. Dans ce cadre, l'objectif de Zope va donc être de publier les objets que l'on construit. Zope déduit d'une demande d'URL la référence à son objet qu'il va retrouver dans sa base de données orientée objet et lui appliquer la requête passée en argument dans l'URL.

Une caractéristique majeure de l'aspect orienté objet construit par Zope est *l'acquisition* qui consiste en fait en une sorte d'héritage contextuel. Un objet hérite des propriétés de l'objet parent ce qui équivaut en pratique au fait qu'un fichier ou dossier hérite du contexte du dossier contenant.

L'architecture du framework en tant que telle est repésentée à la figure 2.15. Le framework est composé de trois sous-composants la base de données d'objets Zope, le serveur en tant que tel et le noyau Zope qui coordonne ces deux composants et les échanges avec les différents systèmes d'enregistrement d'informations. Viennent se greffer sur le noyau Zope des produits Zope (packages en python) qui sont nombreux et fonctionnent un peu comme des plugins pour des applications classiques.

En pratique la plupart des applications web développées tournent généralement avec un serveur web existant comme Apache. Zope fonctionne avec tout serveur web qui respecte le "common gateway interface" (CGI). Pour se passer du ZODB on peut également utiliser directement une base de données relationnelle comme Oracle et PostgreSQL.

2.2.1.2 Les objets Zope

Il n'existe pas, contrairement à Apache de systèmes de fichiers qui contiennent un ensemble des pages et de code source. Tous les objets sont contenus dans la base de données d'objets Zope. L'interface d'admi-

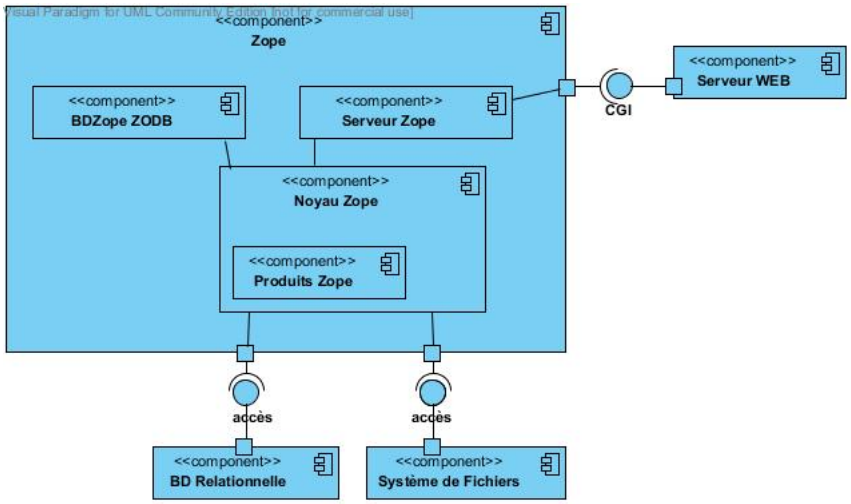


FIG. 2.15 – Composants de Zope

nistration de Zope fait office de navigateur dans la base d'objets Zope et permet d'y effectuer l'ensemble des opérations souhaitées. Les chemins indiqués dans cette navigation correspondent aux chemins menant aux différents objets dans les URL mais ne correspondent pas à un système de fichiers. De ce fait, en dehors de l'interface Zope, on ne sait pas retrouver ces objets. Ils sont en réalité encodés dans un ensemble de fichiers spécifiques.

L'ensemble des objets proposés peuvent être classés dans trois différentes catégories :
- Contenu : cette catégorie contient toutes sortes de données binaires ou textuelles et le fait de travailler avec des objets distants dans une base de données relationnelle.
- Présentation : cette catégorie contient des objets tels que des templates qui permettent de formater le style et la mise en page du site. Zope propose deux services pour faciliter la gestion de la présentation : template de page Zope (ZPT) et des template de documents (DTML).
- Logique : cette catégorie contient l'ensemble des objets impliquant des opérations logiques et abstraites de tout travail de présentation. Ils sont construits au travers de scripts pythons.

Au niveau de la présentation, Zope travaillait dans un premier temps avec DTML pour permettre au moyen d'un système de tags de générer dynamiquement des pages HTML et d'avoir ainsi un affichage paramètrisé. Ils a ensuite travaillé sur ZPT qui visait à pallier certaines failles de DTML. A partir d'un fichier DTML, il est impossible de revoir le code HTML initial dans un éditeur "what you see is what you get". En effet, le format DTML n'est pas un format HTML valide et donc la revue de l'interface en devient contraignante pour un simple graphiste. Dans le cadre de notre interface, c'est le ZPT qui est employé pour définir les pages HTML.

2.2.1.3 Utilisation d'une base de données extérieure

Zope permet l'utilisation d'une base de données relationnelle extérieure au framework. C'est notre cas ici. Il réclame alors l'utilisation d'un adaptateur qui va permettre à Zope de se connecter à la base de données distante. Dans le cas d'Expesurf, il s'agit d'une base de données en PSQL dont l'adaptateur est *Psycopg*. Zope va représenter la base de données distante par un objet de type "database connection". Le framework propose un type d'objet particulier pour exécuter des opérations sur la base de données : les méthodes ZSQL. Une méthode ZSQL va permettre d'éffectuer une séquence d'opérations écrites en ZSQL. L'adaptateur va ensuite interpréter la méthode, adapter le langage au type de base de données connectée et y incorporer les paramètres passés en argument (les méthodes sont paramètrables au moyen de DTML dont les tags sont adaptés pour effectuer toutes sortes d'opérations à la création de la requête). Les résultats de la requête sont présentés sous la forme d'objets Zope "Result object".

Dans ses accès à la base de données, ainsi que dans ses opérations sur sa propre base d'objets, Zope

fonctionne systématiquement avec des transactions. Toutes les modifications des données persistantes (de la base ou des objets) sont effectuées en une fois à l'affichage de la page.

2.2.1.4 Version 3 de Zope

Toutes les informations reprises ci-dessus concernent la deuxième version du framework, c'est à dire celle utilisée par l'interface AC d'Expesurf . La licence particulière que propose Zope permet d'utiliser Zope sans communiquer le code des améliorations apportées au framework. Cette communauté étant plus importante que les autres développeurs de Zope, les failles de la version 2 ont été corrigées de manière indépendante et sans concertation par les différents utilisateurs sans jamais être intégrés à la distribution initiale. Cette maintenance approximative du logiciel ainsi que de réels problèmes conceptuels ont poussé Zope a reprendre à zéro le développement de leur framework. Un grand changement conceptuel apporté à la nouvelle version est l'abandon de la hiérarchie contextuelle qui entraînait des comportements imprévisibles des pages dans les dossiers dépendants. Zope a depuis abandonné toute maintenance de la version 2.

2.2.2 Modélisation de l'interface

2.2.2.1 Le problème de la modélisation d'interfaces

Le modèle UML (unified modeling language) propose un ensemble de types de modélisations principalement conçues afin de modéliser différents aspects des architectures dites orientées objets . A cet égard, UML semblerait particulièrement adapté au cas de figure de Zope. Malheureusement, ce système de modélisation ne propose pas de type de schéma adapté aux interfaces homme/machine. Le modèle propose les use case et scénarii mais qui se veulent de très haut niveau et indépendants de l'interface développée. Dans un contexte de rétro-ingénieurie, ceci s'avère insuffisant puisqu'on doit comprendre l'application jusqu'au niveau de son implémentation (très bas niveau).

De plus si entre le diagramme de composants et les diagrammes de séquence on peut retrouver (pour une même granularité) les même composants et donc avoir aussi bien la vue structurelle qu'interactive du logiciel, c'est n'est pas le cas des scenarii et des schémas use case qui se limitent toujours à une vue interactive de l'interface et peinent à en dégager la structure ou les implications logicielles sous-jacentes (accès à la base de données, modifications de l'environnement etc).

Par ailleurs, si Zope est construit selon une philosophie orientée objet, il travaille beaucoup sur un système de hiérarchie contextuelle avec son concept d'acquisition que UML ne propose pas. Or, c'est le point d'ancrage de l'architecture Zope : construire pour chaque page de l'interface un objet associé, mais surtout, encapsuler ces pages dans des couches hiérarchiques successives. Cette hiérarchie ne sait pas apparaître en UML où le concept le plus proche est l'héritage entre classes qui prêterait à confusion.

Si Zope enregistre en mémoire de façon singulière ses objets, il n'empêche qu'aux yeux du programmeur il propose une vision arborescente de la structure du programme. Cette structure est identique à celle proposée par une navigateur ordinaire de système d'exploitation et il semble intéressant de récupérer cette vision. En revanche, il est impossible dans la représentation Zope de voir à tout moment l'architecture globale du logiciel et donc d'avoir une vue d'ensemble de celui-ci. Comme l'architecture développée l'est uniquement au travers de l'interface, il serait problématique de s'éloigner fortement de cette présentation qui donne la meilleure vue non seulement de l'existant, mais également de ce qu'a voulu construire le développeur. De ce fait, nous présenterons la structure arborescente dans son ensemble en veillant à ne pas omettre d'éléments sémantiques traduits graphiquement par l'interface Zope.

Cette vision statique de l'organisation du logiciel ne permet pas de comprendre l'interaction entre les composants du logiciel et la manière dont on va naviguer au sein de l'interface. Pour ce faire, une deuxième modélisation présentera la découpe depuis la tâche voulue par l'utilisateur en sous-tâches et leurs relations pour présenter la manière dont s'organisent les fenêtres. Cette modélisation se fera selon le modéle des concurrent task trees (CTT) [5].

Les deux modélisations donneront une vision arborescente de l'architecture du logiciel et de la navigation au sein de celui-ci. Le premier va donner une vue partant de la racine du code source et l'autre une vision partant de la première page visitée par l'utilisateur. Les feuilles des deux arborescences se rejoindront puisque la découpe du programme prévoit *in fine* d'arriver à une opération unitaire et qu'une découpe successive en sous-tâches finit par aboutir également à une tâche unitaire, indivisible.

2.2.2.2 Modélisation de la structure du programme

Zope présente l'architecture de son logiciel dans l'interface d'administration de l'application. Cette interface présente un niveau hiérarchique à la fois. Elle distingue les différents objets selon leur type et leur nom identifiant. Il était important de ne pas perdre ces informations dans une modélisation plus générale. La structure arborescente est donc découpée en niveaux représentant tous un étage de la hiérarchie. Des objets de type différent sont représentés par des graphismes différents (figure 2.16). Les flèches sont orientées du contenant vers le contenu et représentent l'association "contient".

Toutes les structures hormis les flèches représentent un objet Zope. L'aspect orienté-objet de l'architecture se retrouve dans les "dossiers". Tous les objets qui ne sont pas de ce type-là peuvent être perçus soit comme des attributs (fichiers de présentation) soit comme des méthodes associées à ces objets (scripts Python). Les flèches représentent le rapport d'acquisition. Le sous-ensemble sur fond bleu est l'ensemble de fichiers et dossiers qui caractérisent tous les objets liés à la base de données : *propriete*, *test*, *critere*, *materiau*, *procede*, *interaction* et *incompatibilite*. Par soucis de clarté nous avons choisi de ne pas tous les représenter. L'architecture de l'interface est très clairement découpée. La découpe rappelle les types d'entités identifiés dans la base de données et on associe à chacun de ces objets de contenu l'ensemble des opérations que propose l'interface : ajouter (*add*), mettre à jour(*update*). Chacun de ces objets est doté d'une présentation unique de son contenu : *display* ce qui assure une bonne modularité de l'interface puisque le même formulaire pourra être utilisé pour toutes les opérations. En revanche, bien que cette structure soit très claire, elle fait apparaître un choix de dissocier complètement les différents types d'entités les uns des autres et n'offrent pas de structure permettant de les mettre en relation. Cette architecture se traduit, dans l'interface, par une présentation des composants distincts sans mettre en valeur leurs relations.

2.2.2.3 Organisation des tâches

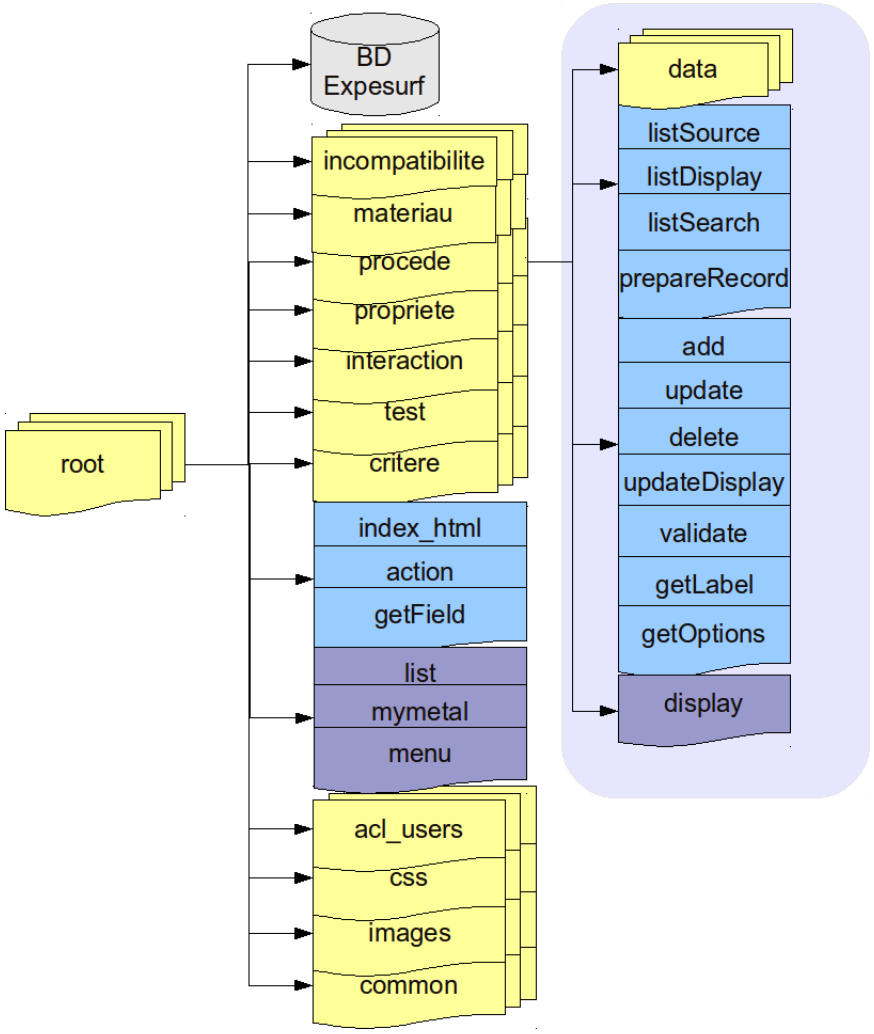
Dans le cadre de notre présentation, il est très important d'avoir une bonne modélisation de l'organisation de l'interface. Cette organisation pourra ensuite être soumise à une critique constructive pour évaluer si la découpe des tâches dans l'interface correspond ou non à l'utilisation qu'on veut en faire. Afin de modéliser cet ensemble de tâches nous allons utiliser la modélisation des Concurrent Task Trees proposée par Paterno [5].

A. Présentation des Concurrent Task Trees

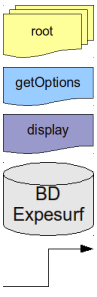
Les Concurrent Task Trees partent du point de vue utilisateur et mettent l'accent sur les tâches qu'il compte effectuer. Et en ce point ils se rapprochent des scenarii proposés par UML. Cependant, le modèle des scenaris ne permet pas de visualiser l'agencement des opérations ni de représenter en une fois les scénarios où une erreur survient ainsi que les scénarii dits "happy" où tout se déroule comme prévu.
Dans le cadre d'une interface chargée de formulaires, la représentation des scenarii peut s'avérer fort verbeuse. Cette représentation part d'une tâche principale racine et fait une découpe de celle-ci en sous-tâches, elles-mêmes découpées en sous tâches et ainsi de suite pour arriver à une arborescence dont les feuilles présentent des tâches unitaires indivisibles. Les sous-tâches peuvent être concurrentes, optionnelles, dépendantes etc. Il est donc important de comprendre la sémantique de la représentation proposée par le modèle CTT afin de cerner précisément les relations entre sous-tâches. Le modèle s'appuie sur sept opérateurs :

- **Activation :** T1 >> T2 ou T1 []>> T2
  après T1 active T2 avec ([]>>) ou sans (>>) passage d'information

- **Désactivation :** T1 [> T2
  l'exécution de T1 désactive l'exécution de T2



Légende :



objet Zope de contenu

objet Zope logique de type script Python

objet Zope de présentation

objet Zope logique de type connexion à une base de données

association liant un contenant (origine) à son contenu.

FIG. 2.16 – Structure de l'implémentation

- **Interruption :** T1 |> T2
  l'exécution de T1 interrompt l'exécution de T2

- **Choix :** T1 [ ] T2
  on exécute T1 ou T2

- **Concurrence :** T1 ||| T2 ou T1 |[]| T2
  T1 et T2 s'effectuent en parallèle avec (|[]|) ou sans (|||) échanges d'informations

- **Itération :** T1* ou T1n

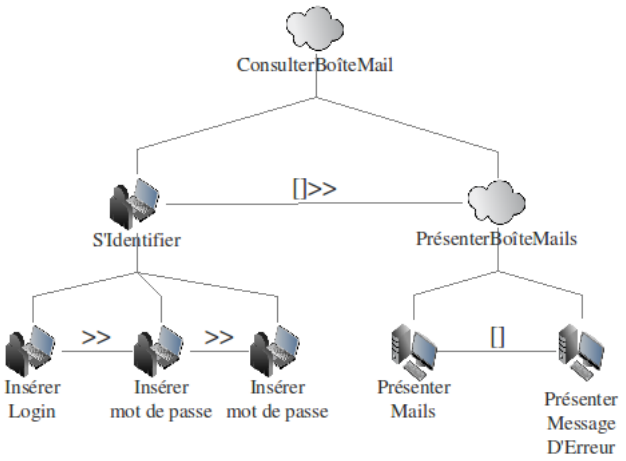


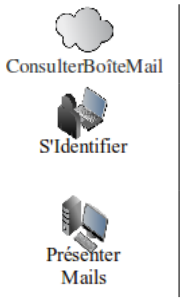
FIG. 2.17 – Exemple de CTT

on répète T1 autant de fois que l'on veut (T1*) ou n fois (T1n)

– **Facultative :** [T]
on peut ou non effectuer T pour atteindre but

Certaines ambiguïtés peuvent apparaître dans la notation et Paternò propose alors d'introduire une priorité des opérations comme suit : [], |||, [>, ». Afin de mieux comprendre l'agencement des opérations principales observons plutôt l'exemple de la figure 2.17 [10].

Il s'agit ici d'une découpe de la tâche globale qui consiste à consulter sa boîte de courrier électronique (*consulterBoiteMail*). Le diagramme indique que pour effectuer cette tâche il faut mener à bien deux tâches successives dont la première est de s'identifier. Les informations à l'issue de cette tâche sont nécessaires pour l'accomplissement de la seconde tâche : "se loguer". S'identifier comprend deux sous-tâches à effectuer séquentiellement aussi : insérer un login et insérer un mot de passe. Ces deux actions ne sont pas interdépendantes [11]. Se loguer peut comporter deux issues différentes et comprend donc deux tâches exclusives : présenter les mails et afficher un message d'erreur. Cet exemple est très basique et se place à un niveau d'abstraction très bas qui ne nécessiterait à priori pas d'analyse aussi poussée mais il permet d'illustrer comment l'arborescence s'organise. De plus, on remarque que les buts (tâches) peuvent être de différentes natures :



Abstration

Interactives (homme-machine)

Applicatives

B. modélisation au moyen des CTT

Nous ne nous placerons pas à un niveau d'abstraction aussi bas pour la modélisation que nous ferons de l'interface. En effet, la complétion des différents champs un à un pour chaque type de formulaire ne présenterait aucun intérêt et compliquerait la représentation. En revanche la découpe fonctionnelle de l'interface pourrait être mise en évidence, abstraite de sa mise en forme et donc être remaniée par la suite de façon plus utile. Le menu propose trois types d'opérations sur les données : des validations, des recherches, et des encodages. Comme la structure de l'interface prévoit d'effectuer ces opérations autant de

[10] L'ensemble des schémas de Concurrent Task Trees a été faite en utilisant les logos de Gliffy (http ://www.gliffy.com) et Open Office Draw.

[11] Une bonne insertion d'un mot de passe ne réclame pas une bonne insertion du login

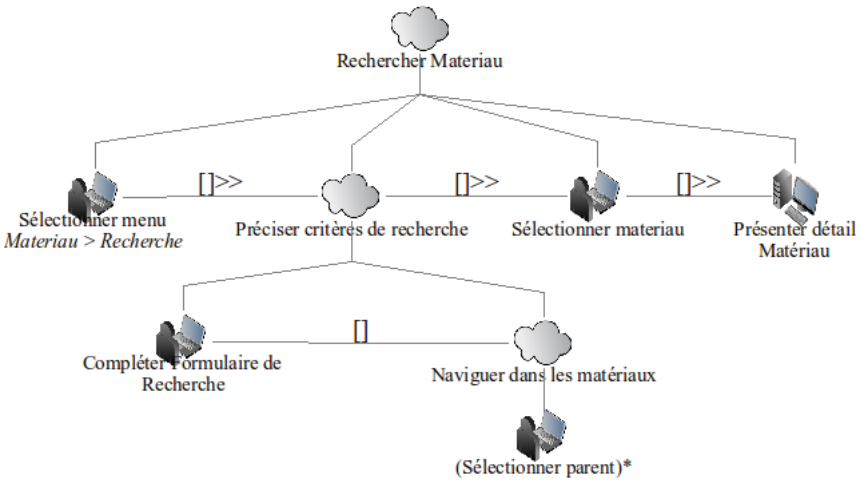


FIG. 2.18 – CTT de recherche

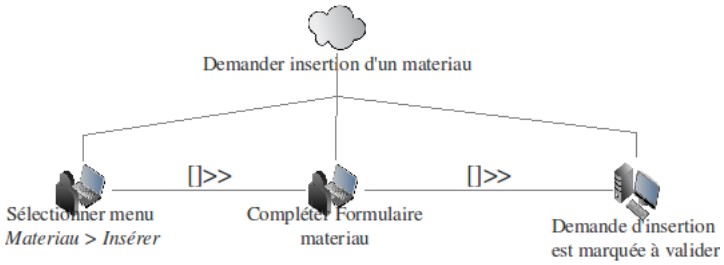


FIG. 2.19 – CTT d'insertion

fois qu'il n'existe de données encodables nous nous contenterons d'une modélisation générique d'encodage et de recherche. L'opération de validation en revanche se fait de manière groupée pour l'ensemble des données.

L'exemple présenté à la figure 2.18 propose une recherche de materiaux mais il est appliquable à n'importe quel type de données. Ici nous remarquons la possibilité d'utiliser deux types de recherche différents. On note aussi, dans le cas de la navigation, un déploiement de profondeur indéfini avant de retrouver la donnée de choix puisque l'on doit parcourir toute la hiérarchie familiale du matériau avant d'y accéder.

Avec le même niveau d'abstraction on peut représenter l'organisation des tâches de l'insertion de données proposée elle aussi pour le cas des matériaux (figure 2.19) mais qui peut s'appliquer à n'importe quel type de données.

La validation (figure 2.20) est une opération particulière du menu puisqu'elle est présentée en une fois pour toutes données confondues à l'inverse de la recherche et de l'insertion pour lesquelles ont doit choisir préalablement le type de données.

Le menu ne propose pas une vue exhaustive des opérations qui peuvent être effectuées. En effet, il est possible d'éditer (figure 2.21) et de supprimer (figure 2.22) des données. Dans les deux cas, on recherche la donnée dans un premier temps. On sélectionne ensuite l'édition ou la suppression. L'édition mène au formulaire d'insertion et donc les opérations de suppression et d'édition sont une combinaison de sous-tâches des tâches présentées auparavant.

Ici encore, l'exemple donné concerne le menu matériau mais est appliquable à tous les autres menus et sous menus. Ces deux opérations étant dépendantes de la recherche, leur efficacité dépendra beaucoup de celle de la recherche. C'est donc une tâche dont l'implémentation est à soigner tout particulièrement.

Bien que la sémantique des deux représentations ne soit pas la même, il est intéressant de noter le parallèlisme entre l'architecture de l'implémentation de l'interface et la découpe proposée par les Concurrent Task Trees. En effet, l'organisation hiérarchique part dans les deux cas d'un choix initial du type de donnée concerné pour descendre vers un affichage des informations détaillant l'objet dans un formulaire.

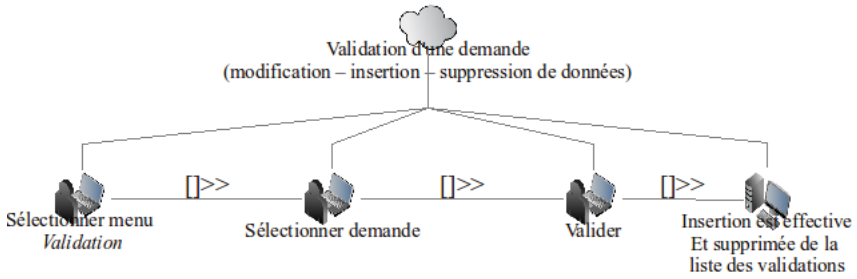


FIG. 2.20 – CTT de validation

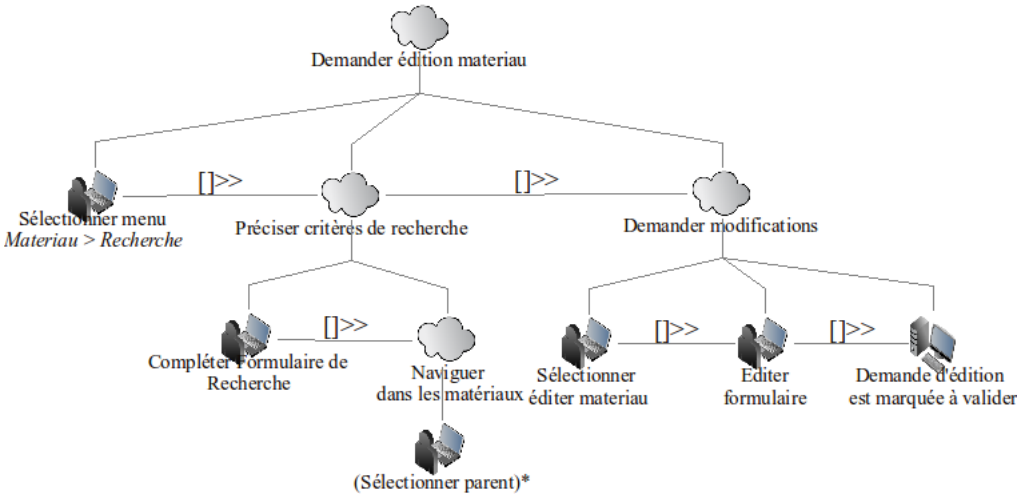


FIG. 2.21 – CTT d'édition

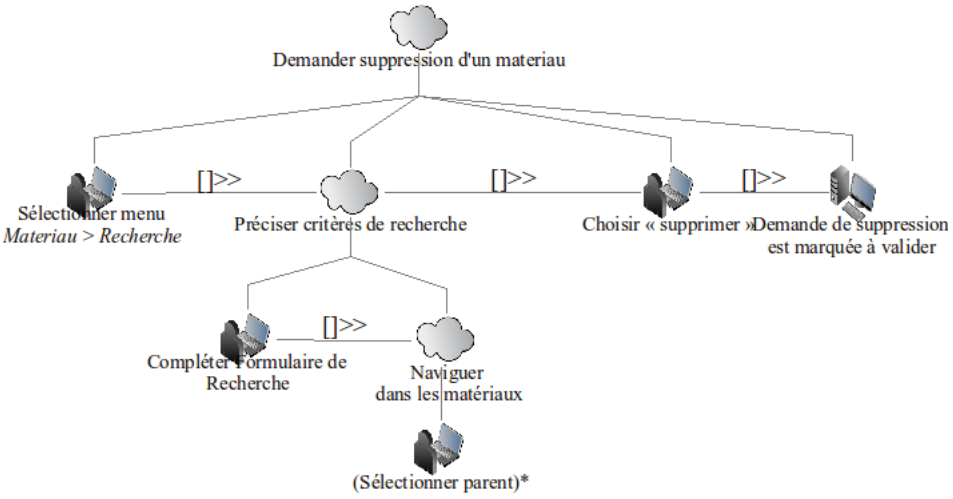


FIG. 2.22 – CTT de suppression

2.3 Conclusions

Au travers de la rétroingénierie, d'une part, et d'entretiens avec les utilisateurs du système, d'autre part, nous pouvons dégager un ensemble de problèmes et d'atouts de l'existant. Les trois composantes de l'application que nous avons vues (Zope, l'interface utilisateur, la base de données) présentent toutes des caractéristiques propres qui sont sources de problèmes. Il est important de cerner à ce stade déjà, où se situent les point faibles et forts de l'application existante et ce afin de concentrer les efforts de l'analyse des exigences autour de la correction de ces difficultés. Au travers de cette première analyse nous avons abordé trois points différents de l'application : la base de données, l'implémentation en Zope de l'interface d'acquisition de données, et l'organisation de cette interface en elle-même. Ces trois points sont également la source de problèmes très distincts et sur lesquels nous avons une emprise bien différente en fonction des interdépendances avec le reste du programme.

2.3.1 Au niveau de la base de données

Au niveau de la base de données, un travail important sur des triggers a été effectué pour permettre un meilleur contrôle de l'encodage. Ce contrôle se présente maintenant comme une caractéristique plus contraignante qu'avantageuse de l'application et c'est un premier point qu'il serait intéressant de contourner. Au delà du fait qu'elles sortiraient du cadre de ce mémoire, des modifications de la base de données ne seraient pas souhaitables à plus long terme. En effet, il n'est pas impossible, qu'à l'avenir, pour certains types d'utilisateurs, on ne souhaitera pas réintroduire ce système de double vérification. De ce fait, il est intéressant de conserver ce type de fonctionnalité que l'on peut aisément contourner. Elle permettra à l'utilisateur expérimenté du système un encodage sans validation et donc plus rapide.

La base de données présente également une structure familiale pour certaines de ses tables comme on a pu le remarquer dans le schéma entité-association. Cette structure pose problème au niveau de l'interface utilisateur parce qu'elle ralentit deux opérations clés : la recherche et l'insertion. Si l'utilisation de ce système d'héritage était prévu initialement pour ne pas démultiplier un encodage de données (les propriétés s'appliquent automatiquement à un ensemble de matériaux par exemple) il s'avère que dans la pratique les propriétés familiales sont peu employées et qu'elles en viennent à encombrer la base de données et l'utilisation de l'interface. Encore une fois, cette structure est une richesse de la base de données qu'il est important de conserver quand bien même on ne s'en servirait pas. A l'inverse d'une interface qui évolue régulièrement en fonction de l'évolution des technologies et des demandes des utilisateurs, il est impotant de souligner ici qu'une base de données est une structure stable dans le temps. De plus, la base de données ici est le noyau central de l'application dont tous les modules (interfaces, moteur d'inférence) dépendent. Son bon fonctionnement est essentiel et un changement sur celle-ci risquerait d'avoir répercussions sur le reste de l'application.

2.3.2 Une implémentation en Zope

L'implémentation en Zope présente trois inconvénients majeurs. Le premier inconvénient, assez évident est l'organisation même de Zope qui impose un univers de développement particulier dans lequel tout programmeur voulant travailler l'interface devra se plonger. Dans cette organisation orientée objet les opérations ne sont pas divisées de manière fonctionnelle comme dans le design pattern model - view - controller . De ce fait, un graphiste va devoir travailler à travers tout le code dans des dossiers comprenant de la programmation et de la mise en page. L'organisation de Zope impose aussi ses langages de programmation et outils dont découle un certain manque de flexibilité.

La deuxième difficulté de ce framework est qu'il n'évolue plus. La maintenance et le développement de Zope2 ayant été interrompus, l'adaptateur pour la base de données (voir figure 2.15) n'est plus à jour avec la dernière version de PostgreSQL. De ce fait, une mise à jour de la base de données la rendrait inaccessible pour l'interface d'acquisition de connaissances. Du fait même de la nature du problème, continuer à travailler avec Zope et tenter de contourner cet embarras est inenvisageable. Toutes les opérations proposées par cette interface devront faire l'objet d'un nouveau développement complet.

Finalement, Zope2 utilise des morceaux de Javascript qui ne fonctionnent pas correctement sous Internet Explorer. Or, il s'agit d'un des deux navigateurs les plus répandus et il est important que l'application soit portable.

2.3.3 L'interface utilisateur

De l'implémentation et de la base de données découlent des problèmes d'interface qu'il est important de noter. L'interface utilisateur est structurée de manière très claire mais un peu trop découpée. En effet, le fait de faire opérer l'utilisateur en fonction du type de données auquel il s'adresse empêche la mise en évidence des relations entre celles-ci. Et pourtant, c'est un des grands intérêts de cette application : faire la relation entre matériaux, propriétés, tests, applications, etc. En plus de ce choix initial contraignant, les opérations proposées ne répondent pas non plus aux besoins de l'utilisateur, à savoir une recherche facile et la possibilité de faire de nombreuses insertions. Les utilisateurs sont capables avec l'interface actuelle de faire trois à quatre encodages par heure maximum pour les incompatibilités alors qu'il peut leur arriver de devoir en effectuer des centaines. La longueur des formulaires fait la richesse de la base de données parce que le détail des informations encodées va permettre d'obtenir des solutions plus fines. Il est difficilement concevable de ne pas perdre l'essence du logiciel en omettant des champs. En revanche, la hiérarchie familiale qui encombre la recherche et allonge les formulaire pourrait être aplatie. On note alors que l'encodage de propriétés groupées va devoir être facilité puisqu'il deviendra plus fréquent. Une attention toute particulière devra être apportée pendant la phase d'ingénierie des exigences pour cerner la façon dont ces opérations vont évoluer. Il faut actuellement effectuer deux tâches principales (encodage et validation) pour effectuer une insertion. Il est possible de travailler sur la réduction du nombre de sous-tâches (CTT) et de contourner la validation des données en validant par programmation les nouvelles insertions au fur et à mesure.

La recherche se verrait également simplifiée par de telles opérations. La navigation dans les matériaux permet actuellement de visualiser leur héritage familial mais ne permet pas de voir toutes les informations qui dépendent d'un matériau donné. Or, comparativement, les propriétés, nomenclatures, ... d'un matériau comportent un contenu plus intéressant que la seule hiérarchie familiale du matériau.

Si l'interface présente certains points d'ombre elle a des avantages qu'il sera important de conserver dans la nouvelle application. En effet, s'il est important de comprendre ce que la nouvelle interface devra corriger, il est aussi utile de savoir mettre en évidence les atouts de l'interface existante qui devront se retrouver dans l'interface finale.

L'interface a une organisation claire. Il est facile à tout moment de s'orienter et de savoir où on est. Une organisation et une structure simple et intuitive sont essentielles pour une utilisation simplifiée de l'application. Il est intéressant de noter que les hiérarchies de tâches sont généralement courtes et cela témoigne de la facilité de s'orienter dans les menus. Depuis la page d'accueil il ne faut pas beaucoup de clics pour arriver en tout point de l'interface.

Un autre avantage est le fait qu'il existe plusieurs manières d'accéder à une fonctionnalité (deux modes de recherche, différentes manières d'accéder à l'édition d'un composant, etc.). Grâce à ces nombreux accès pour une même fonctionnalité de l'application, les chances que l'utilisateur trouve rapidement ce qu'il veut augmentent.

La nomenclature et les repères visuels de l'interface sont aussi forts simples. On ne doute jamais de l'opération qu'on s'apprête à effectuer en sélectionnant un bouton ou un menu.

En conclusion, la clarté de la nomenclature, des icônes, et du plan du site sont bons. Le fait de dupliquer les manières d'effectuer une opération rend l'interface plus facile d'utilisation. Ces éléments sont primordiaux et il sera important de les conserver. En revanche, les modes de recherches ne sont pas suffisamment flexibles et les résultats ne sont pas présentés de façon suffisamment lisible et complète. De son côté, l'opération d'encodage est trop fastidieuse.

Chapitre 3

Analyse des exigences

3.1 Analyse préliminaire

La phase d'analyse des exigences est toujours un travail conjoint du développeur et du client. Tout ce chapitre et ses conclusions sont donc le fruit d'un travail avec le «client». C'est pourquoi, avant toute chose, il est important de cerner à qui l'on s'adresse et à quel contexte d'utilisation et type d'utilisateur se destine l'interface.

Dans un premier temps, le client et l'utilisateur se confondent. Le projet étant encore en développement aussi bien au niveau du projet informatique que de l'expertise métallurgique, les utilisateurs du logiciel sont les clients. Par la suite, il est probable que ce type de logiciel sera certainement manié par un expert. On sera rarement dans un contexte d'utilisation parallèle de nombreux utilisateurs. L'outil complet (Expesurf) pourra être déployé chez le client qui enrichira lui-même sa base de données ou au travers d'un serveur centralisé où différents utilisateurs accèderaient au service grâce au web. Notre interlocuteur représente donc l'ensemble des clients et utilisateurs : l'équipe de recherche du projet Expesurf.

Développer une application pour la deuxième fois n'est pas le même exercice que de démarrer un développement à partir de rien. On va pouvoir mettre en évidence certains défauts de l'interface existante au profit d'un second développement en visant à atteindre un meilleur produit. En revanche, on devra adresser la difficulté de travailler avec des utilisateurs ayant développé des habitudes spécifiques face à ce logiciel : des habitudes pour contourner son fonctionnement (insertion manuelle d'informations dans la base de données, contre-validation personnelle des informations) et d'autres liées simplement à l'utilisation courante du logiciel. Il est donc d'autant plus difficile de ne pas brusquer l'utilisateur en proposant une interface tellement novatrice qu'elle va dérouter.

On court le risque contraire également. Après un long exercice de rétro-ingénierie et d'utilisation de l'interface existante, on peut tenter de partir d'hypothèses de développement liées à l'existant qui mériteraient d'être remises en question. Pour éviter ces difficultés, nous utiliserons les Concurrent Task Trees. Ils se placent à un niveau d'abstraction tel, que leur utilisation va permettre de s'abstraire de l'existant tout en ne perdant pas le fil conducteur de l'application. Ces modélisations délimiteront également l'ensemble des exigences fonctionnelles de l'interface.

En réalité, comprendre les attentes de l'utilisateur se fait par affinage progressif et, avant d'arriver à une découpe des tâches de type CTT ou à une proposition d'interface, il faut avoir isolé un certain nombre d'informations. Au travers de la rétro-ingénierie effectuée et des différentes rencontres avec le client, des informations générales comme le contexte d'utilisation de l'application et les opérations qu'elle doit effectuer ont pu être mis en évidence. C'est au moyen de ces informations (décrites dans une première partie) que l'on va pouvoir élaborer une organisation des tâches.

Cette organisation des tâches va permettre de proposer une première structure des interfaces au travers de croquis sur papier que l'on va pouvoir adapter avec le client. Au travers de cette démarche nous pourrons mettre en évidence les éventuels oublis de la première analyse et la corriger. Nous aurons également une vue plus concrête des attentes graphiques de l'utilisateur[1].

Enfin, la discussion autour des croquis va déboucher sur un prototypage. Cette démarche de remise au propre des discussions sur les interfaces va donner une ligne directrice au développeur. Ces prototypes

[1] Dans le cadre de ce développement, les utilisateurs sont les «clients».

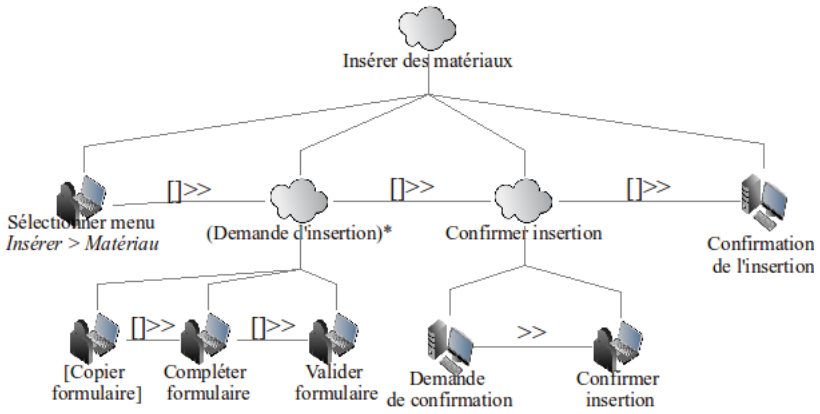


FIG. 3.1 – CTT de l'insertion groupée de matériaux

ne définissent en aucun cas les interfaces définitives et constituent davantage un cadre pour démarrer le développement plutôt qu'une fin en soi. Ainsi ce premier objectif va être réévalué au fur et à mesure du développement en fonction des réalités pratiques que les prototypes ne prendraient pas en compte.

Après avoir établi un bilan des fonctionnalités de l'interface existante, nous présenterons les modélisations CTT que nous avons initialement établies sur base des modélisations précédentes et d'une critique de celles-ci. Ensuite, nous présenterons le prototype accepté par l'utilisateur.

3.1.1 Ensemble des opérations de l'interface

L'avantage de ce travail est de pouvoir partir de l'existant. La première interface répondait déjà à une analyse des exigences. De plus, la critique qui en a été faite permet de distinguer les parties qui manquent aux exigences de celles qui les remplissent. L'interface actuelle propose l'édition, l'insertion, la suppression et la recherche de six types de données différentes correspondant chacune à une table spécifique de la base de données : incompatibilité, matériau, interaction, procédé, critère, propriété et test. Parmi ces différentes insertions il resort que certaines sont moins importantes que d'autres et pourraient être intégrées plus tard dans l'interface. L'insertion, l'édition ou la suppression d'un matériau gèrent également ces opérations pour les classifications et les applications qui la concernent. L'interface actuelle permet et impose une validation des modifications demandées. Cette fonctionnalité doit disparaître. En revanche, la recherche ne doit plus se contenter de présenter les informations des différentes tables de manière isolée mais de mettre en valeur les relations entre elles. On va effectuer des insertions groupées afin de limiter le réencodage de certains champs. Ces deux opérations s'étendent et prennent donc un caractère plus critique.

3.1.2 Modélisation des tâches

Les tâches principales vont toujours partir de l'insertion et de la recherche. Après avoir sélectionné l'insertion, on sélectionne l'objet auquel elle se rapporte. C'est cohérent avec la démarche de la recherche où on ne peut présenter plusieurs types d'informations en même temps (voir figure 3.1).

Une première différence essentielle est qu'on n'introduit plus une demande d'insertion mais bien une insertion à part entière. L'insertion dans l'interface précédente réclamait deux opérations : une demande d'insertion et une confirmation de l'insertion. Du fait du besoin de validation, tâche indépendante, l'interface ne proposait pas la tâche d'insertion à part entière. La demande d'insertion ici passe par un formulaire et la possibilité de le valider. Cette opération peut être effectuée un nombre indéfini de fois avant de confirmer l'insertion. La confirmation rend l'insertion définitive. Les demandes d'insertion successives peuvent ou non faire intervenir la copie d'un encodage existant qui préremplit le formulaire.

La recherche implique un choix initial de l'objet de la recherche : un matériau, un procédé, une incompatibilité, ... (voir figure 3.2). Ensuite, vient la complétion du formulaire de recherche associé. Finalement, on valide son choix. Les résultats de la recherche apparaissent. La recherche immédiate comme celle-là est simple mais elle se rapproche de celle de l'interface initiale. Elle ne permet pas de visualiser l'association entre les matériaux trouvés et un ensemble de propriétés qui les caractérisent. Elle ne permet pas non plus de savoir quelles sont les incompatibilités dont un matériau fait l'objet. C'est

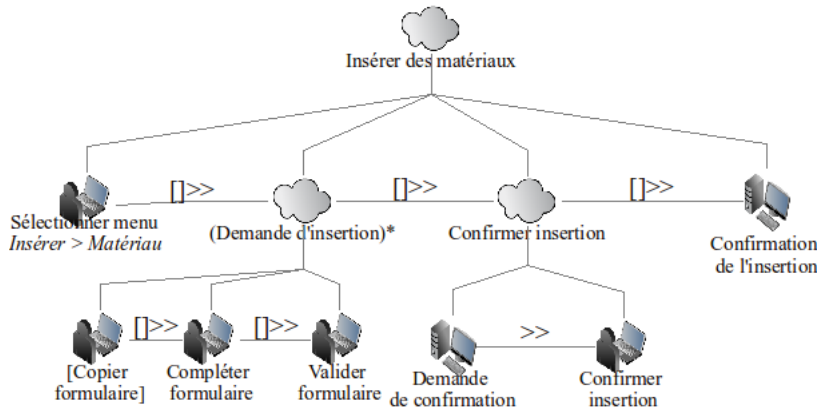


FIG. 3.2 – CTT de la recherche de matériaux

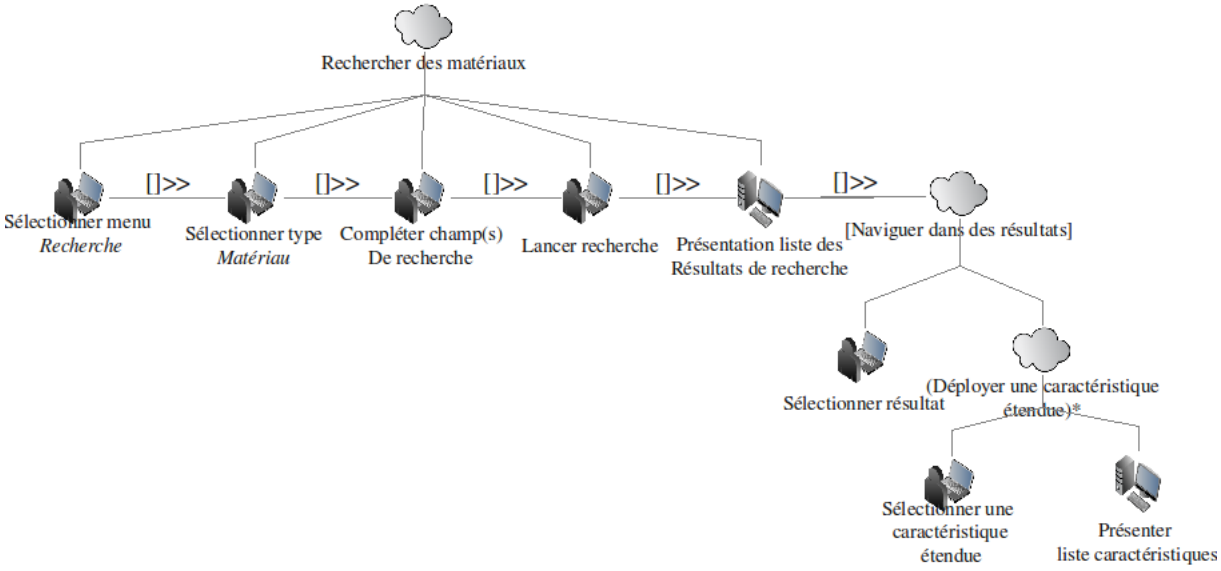


FIG. 3.3 – CTT de la recherche de matériaux avec navigation

pourquoi, il faut pouvoir naviguer au sein des résultats de recherche. Une question reste sans réponse : comment présenter les résultats de la recherche ? Il serait en effet intéressant, une fois un ensemble de matériaux délimités, de pouvoir parcourir leurs différentes caractéristiques. La recherche directe devient alors la première étape d'une recherche navigationnelle (voir figure 3.3.

3.2 Maquettes

Après une analyse de l'organisation des tâches, il était important de proposer une maquette des interfaces afin de discuter plus concrètement du mode d'accès à ces fonctionnalités et de leur organisation. En partant de brouillons, nous avons construit les boutons et les menus qui permettraient d'effectuer les tâches définies précédemment.

Ces maquettes ont donné lieu à des prototypes dits «Lo-fi» [2]. Ils sont tout simplement une remise au propre des prototypes dessinés avec le client. Ceux-ci se distinguent des prototypes «Hi-fi» qui sont déjà un prédéveloppement des interfaces et présentent déjà (avec des résultats et données fictifs) les interactions de l'interface. Nous nous sommes contentés de prototypes «papier», indépendants de la technologie utilisée et donc davantage réutilisables. Les prototypes Hi-fi sont plus utiles pour des tests plus avancés avec un grand nombre d'utilisateurs, ce qui n'est pas le cas de notre développement.

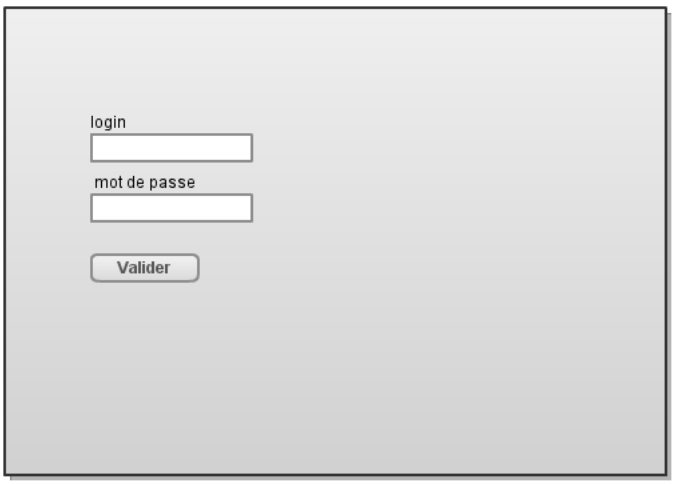


FIG. 3.4 – Login

Ces prototypes permettent de mettre en forme les attentes avec un graphisme plus proche de la réalité. On y visualise clairement les attentes graphiques en plus des attentes fonctionnelles définies préalablement au travers des CTT.

Bien que dans le courant du développement certaines exigences ont disparu au profit d'autres il est intéressant de présenter l'ensemble des attentes concrêtes du client lors de notre première rencontre et de comprendre comment et pourquoi elles ont évolué vers l'interface que vous découvrirez par la suite.

3.2.1 Login et page d'accueil

L'utilisateur arrive en premier lieu sur le formulaire d'authentification (voir figure 3.4) où il complète son nom d'utilisateur et son mot de passe. A ce propos, il convient d'introduire deux niveaux d'utilisateurs : un utilisateur avec des droits purement consultatifs, et un utilisateur administrateur qui peut consulter, éditer, supprimer et encoder des données. On confie également à ce dernier utilisateur la gestion des autres comptes utilisateurs. [2]

Une identification réussie amène ensuite l'utilisateur à la page d'accueil (voir figure 3.5) présentant un menu en deux parties : insertion de données (accessible uniquement à l'utilisateur administrateur, grisé sinon) et consultation de données. Le menu sera horizontal plutôt que vertical afin de gagner sur l'espace en largeur vu que l'on fait appel à des indentations pour représenter des hiérarchies d'informations (voir figure 3.7). L'insertion présente des sous-menus liés aux types d'insertions souhaités. Il a été limité aux catégories présentées sur l'exemple de page d'accueil (figure 3.5) du fait que les autres insertions rarissimes et ne nécessitent pas une interface d'insertion[3]. Certaines de ces insertions cachent néanmoins des insertions imbriquées. Ainsi, le formulaire d'insertion d'un matériau peut impliquer l'insertion de ses propriétés, de ses différentes dénominations (classifications) et de ses applications selon sa nature : couche, substrat ou les deux.

3.2.2 Insertion de données

Une fois l'insertion d'un type de données sélectionné on aboutit sur la page d'insertion (voir figure 3.6).

Celle-ci est divisée en deux parties : en haut les lignes encodées jusque là qui attendent à être insérées, en bas le formulaire d'insertion d'une ligne. L'insertion n'est effective qu'après avoir appuyé sur le bouton "insérer" au pied des lignes à insérer. Une fenêtre de confirmation apparaît alors en popup et à la confirmation les informations seront définitivement insérées dans la base de données. On note

[2]Toutes les représentations d'interfaces présentées dans cette section ont été faits au moyen de l'outil Gliffy disponible sur `http://www.gliffy.com`

[3]En effet, le client estime pouvoir insérer ces éléments au cas pas cas dans la base de données au travers de l'interface d'administration de celle-ci (une interface *phppgadmin*).

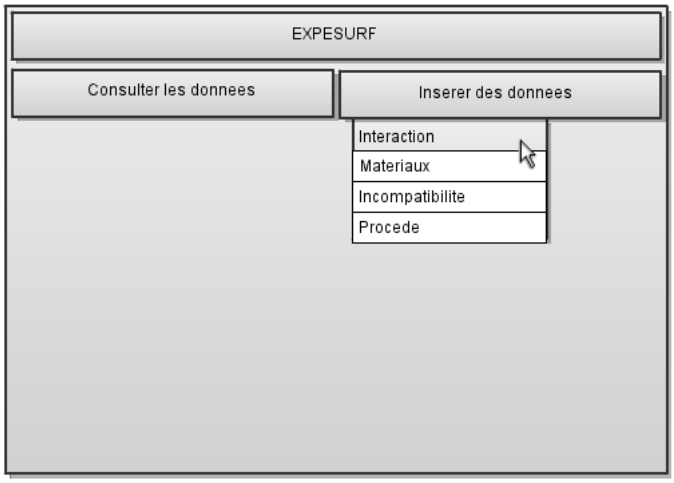


FIG. 3.5 – Accueil

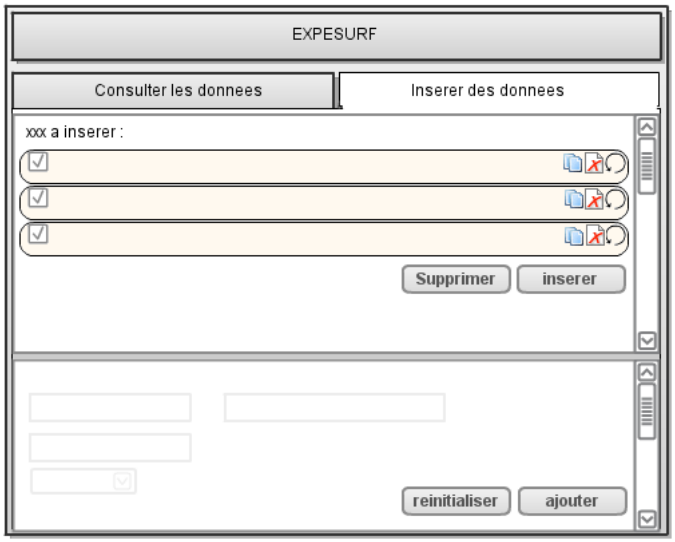


FIG. 3.6 – Onglet d'insertion

que dorénavant il n'y aura plus de contre-validation par un tiers. Dans ces schémas, la flèche représente l'édition, la feuille avec une croix la suppression, et deux pages superposées la copie des données ce qui auto-complète le formulaire d'insertion avec les données de la ligne concernée et permet ainsi un encodage plus rapide (deux clics par ré-encodage) pour les informations redondantes telles que les incompatibilités. En effet, nous avions conclu que nous ne pouvions pas les insérer en masse car les solutions sont différentes d'une incompatibilité à l'autre. La possibilité de copier une incompatibilité afin d'éviter des réencodages et de ne modifier que les parties nécessaires permettra un gain de temps. Les formulaires comprendront donc les mêmes champs mais feront abstraction de la relation familiale. En choisissant la fonction édition (flèche arrondie) sur une ligne, ses champs deviendront éditables dans le formulaire du bas et le bouton "valider" deviendra "éditer". Le bouton "reinitialiser" revient à un formulaire vide d'ajout. Si on était en train d'éditer une ligne, l'édition de cette ligne est annulée. Les encodages, éditions et suppressions ne réclament pas de confirmation puisqu'elles n'agissent pas sur la base de données. Les modifications d'un ensemble d'encodages temporaires sont donc rapides. La validation de l'insertion se fait au travers du bouton "valider" qui, lui, rend les modifications définitives. L'ensemble de lignes se vide alors avec confirmation, ou non, de la bonne insertion des encodages validés.

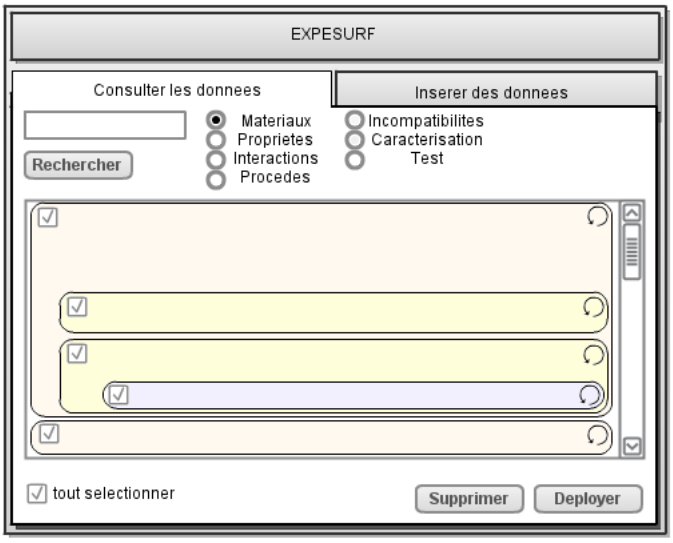


FIG. 3.7 – Onglet de recherche

3.2.3 Consultation des données

La consultation des données a fait l'objet d'une attention particulière. Une première analyse du problème revenait à abandonner le modèle de recherche au travers de formulaires spécifiques au profit d'une recherche sur base d'un seul mot clé. L'utilisateur sélectionnerait ensuite le type d'entité sur lequel faire porter la recherche. On associerait à chaque type d'entité un ensemble de champs clés concernés par la recherche (voir figure 3.7).

Sur l'interface de recherche, des boutons dits *radio* permettent de sélectionner l'objet de la recherche. Les résultats sont présentés sous la forme d'une liste de lignes. Chaque ligne représente un résultat répondant au critère introduit : le terme ou l'expression apparaît dans n'importe quel champ de la table (ou ensemble de tables) contenant les informations liées à cette recherche. Initialement, il était question que cette recherche porte sur un ensemble de champs des types d'entités concernés. Finalement, nous avons opté pour un système de formulaire de recherche par type d'entité. Une ligne de résultats est composée d'un sous-ensemble des informations de l'objet concerné. Ce sous-ensemble comprend les champs qui servent déjà dans la représentation des lignes de résultats dans l'interface actuelle (nom, numéro, ...). En la sélectionnant, la ligne s'agrandit pour former un bloc contenant toutes les autres informations propres à cet objet. Des sous-blocs qui se déploient en cliquant sur des liens dans le bloc parent permettent de visualiser les informations associées au bloc parent.

Cette vision a été pauffinée au travers de nouvelles entrevues. De plus, il a été déterminé que les utilisateurs feraient régulièrement le même type de recherche. Il semble donc pertinent de leur permettre d'enregistrer des requêtes afin de pouvoir les réexécuter sans recompléter le formulaire de recherche.

Une autre question que l'on s'est posée était de savoir s'il était intéressant de limiter la profondeur des déploiements afin que la représentation ne devienne pas confuse pour l'utilisateur. Mais elle n'a pas été retenue. Ainsi, l'utilisateur peut partir d'un matériau donné, déployer ses interactions et se déployer de nouveau. Visuellement il serait dont imbriqué dans sa première représentation. Cela ne semble pas poser de problème et l'utilisateur pourra donc naviguer de manière récursive d'un point à un autre du graphe en déployant indéfiniment une chaîne de cases imbriquées (voir figure 3.7).

Pour l'utilisateur administrateur, la recherche permet de présenter des éléments qu'il a le droit de supprimer et d'éditer. En choisissant l'édition, un formulaire pré-rempli apparaîtra dans un pop-up et permettra de modifier les informations. La suppression et l'édition ouvriront une fenêtre de confirmation. Après la confirmation la modification sera définitive. Il n'existera donc plus de seconde validation. La case "sélectionner tout" aurait permis d'effectuer l'opération de suppression ou de déploiement de façon groupée. Ainsi on aurait pu obtenir en 2 clics l'affichage d'un détail de toutes les lignes. Une seconde sélection de tout suivi de déployer déploie le niveau suivant de tous les blocs/lignes et ainsi de suite. Par la suite, cette fonctionnalité ne sera pas maintenue. Dans le courant du développement il s'avère que le

déploiement de nombreux blocs sur une même fenêtre est visuellement déroutant. Il n'est pas inintéressant qu'il se fasse progressivement plutôt qu'en une seule fois.

3.2.4 Gestion des utilisateurs

Il serait intéressant de proposer à tout utilisateur de niveau "administrateur" une interface basique de gestion des comptes utilisateurs. Il permettrait de créer, d'éditer ou de supprimer des comptes utilisateurs. Cette partie moins déterminante de l'interface n'a pas fait l'objet d'une maquette. Par contre, elle devra s'assurer d'être simple d'utilisation car un repproche fait au système en place actuellement est la complexité de la gestion des comptes utilisateurs pour le système complet.

3.3 Conclusions

Ces maquettes étaient volontairement définies au travers de schémas abstraits de toute technologie. Leur but est de véritablement définir des modèles d'interfaces à implémenter et de s'assurer la bonne compréhension des attentes du client par le développeur. Au travers des fonctionnalités qu'elles requièrent, elles vont permettre de guider les premiers choix de développement et représenter l'objectif à atteindre du point de vue des interfaces. Si les schémas CTT ont permis d'élaborer les premiers jets des maquettes, ils ne sont pas à abandonner. Ils représentent également un produit de l'analyse des exigences. Le niveau d'abstraction des CTT est tel que, malgré des changements de l'allure de l'interface, si l'analyse représente bien les attentes de l'utilisateur, l'interface finale respectera rigoureusement les diagrammes.

Chapitre 4

Développement

Après avoir confronté les attentes de l'utilisateur avec les moyens de développements nous sommes arrivés à un consensus concernant l'interface à développer. La difficulté suivante consiste à déterminer quel chemin parcourir pour atteindre ces exigences. C'est ce processus que ce chapitre va décrire.

Définir comment aborder un développement c'est répondre à des questions technologiques et d'architecture. Ces problèmes sont interdépendants et trancher en faveur d'une technologie délimite les choix d'architecture possibles et inversément. Nous nous souvenons de l'architecture très découpée définie par Zope qui avait conduit à une interface où les relations entre les différents types d'entités n'étaient plus mises en valeur. A l'inverse, s'il avait été décidé de mettre en valeur les relations entre les différents composants, l'architecture adaptée à ce développement aurait peut-être perdu en clarté. Elle aurait peut-être même été inadaptée au framework. C'est pour cette raison qu'il est important de déterminer quel choix (architectural ou technologique) primera sur l'autre et comment les concilier.

Dans le cadre de ce développement la première décision a porté sur un choix technologique. Les attentes du client à ce propos ont été déterminantes. Une réponse à des questions de dynamisme et de portabilité de l'application ne peuvent être résolues qu'au travers du choix judicieux d'une technologie en particulier. Dans le cadre de ce projet, l'architecture s'est donc pliée à un univers de développement défini par le framework Flex.

Le deuxième objectif de l'application a été de miser sur une architecture aussi claire que possible et de s'y tenir avec rigueur. Cette priorité découle directement du fait que ce développement n'est pas un développement final. Il fera encore certainement l'objet de modifications futures d'où l'importance d'une grande lisibilité du code et d'une bonne modularité des différents composants.

Le choix d'une technologie et d'une architecture suffisent à aborder le développement du logiciel et de s'approprier le travail fini dans l'ensemble. Il reste cependant quelques zones d'ombre. A différents moments du développement, on peut se retrouver avec des points de choix dans des parties plus critiques de l'application. Il est important d'expliquer les options qui se présentaient et de justifier les décisions prises. Cette démarche permettra à un futur développeur de cerner plus rapidement l'intérêt et les implications de ces choix qui ne paraissent pas toujours clairs à postériori et qui, dans le cas contraire, pourraient mener à des erreurs.

Ce chapitre vise donc à documenter la succession des décisions technologiques, architecturales et d'implémentation qui ont motivé le développement de l'interface. Nous le cloturerons enfin par une présentation des interfaces obtenues.

4.1 Choix des technologies et le modèle MVC

4.1.1 Applications Web

Jusqu'à récemment, le modèle de développement de sites internet était d'une grande homogénéité. Il impliquait l'utilisation d'un langage de présentation unique : HTML ou, plus récemment, XHTML. Afin d'adapter les pages à un contenu de bases de données, elles pouvaient être développées dans un langage de programmation tel que PHP qui était ensuite interprété en HTML à chaque demande de l'utilisateur final (voir figure 4.1). Ce couplage proposait une présentation statique (fichiers HTML figés) de données variables qui pouvaient être personnalisées au travers de PHP et d'une base de données.

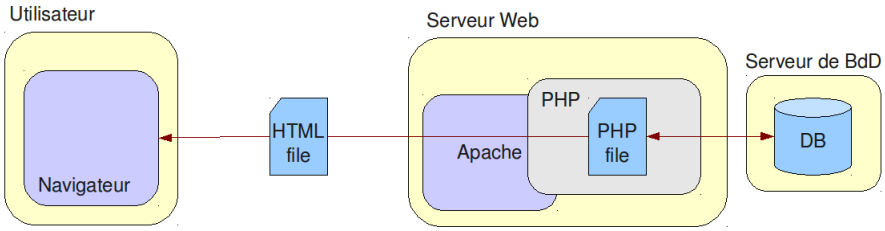


FIG. 4.1 – Développement web statique classique

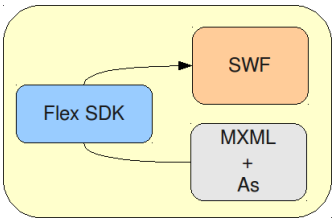


FIG. 4.2 – Compiler Flex

Cette adaptation possible de la page au contenu, est déjà une avancée considérable par rapport au web initial mais l'utilisateur actuel attend un dynamisme de l'interface tout autre. L'incorporation de nouveaux langages exécutables à même le client et capables de modifier la présentation ont fait leur apparition. Ils permettent à l'interface de l'utilisateur de se modifier elle-même : redimensionnements, apparition/disparition de composants, visualisation de contenu multimédia etc. Ils permettent également de faire des appels ponctuels au serveur pour réclamer de nouvelles données à intégrer à la présentation.

Parmi ces langages, on compte notamment Javascript qui a joui d'un grand succès dès ses début mais a vite fait preuve de grandes difficultés de compatibilité. En effet, le langage, et sa compilation complètement libres ont laissé trop de libertés aux navigateurs qui se sont permis d'en faire des interprétations différentes. En particulier, Internet Explorer exécute certaines instructions Javascript différemment de ses concurrents. Le navigateur occupe une position dominante et le langage a du s'adapter au navigateur plutôt que l'inverse. De ce fait, tous les développements Javascript doivent prendre en compte leur environnement d'exécution et exécuter un jeu d'instructions différent en fonction de leur hôte.

Un autre langage propose une alternative puissante et incontournable à Javascript : Flash. Adobe propose dès 1997 un lecteur à télécharger et qui s'intègre au navigateur (*plug-in*) de l'utilisateur pour lire les fichiers *.swf* dits "flash". Le fichier swf est chargé par le navigateur du client à la manière d'une image ou de n'importe quel autre fichier. Le fichier est alors compilé sur la machine du client par le lecteur flash qui l'exécute ensuite. Ces fichiers peuvent s'intégrer à une présentation HTML et présentent l'énorme avantage, comme ils sont compilés à même l'hôte, d'accéder à l'API de son système d'exploitation et donc d'utiliser les fonctionnalités de son installation : webcam, carte son/vidéo, ect. C'est cette fonctionnalité et la possibilité d'intégrer des éléments dynamiques qui va contribuer considérablement au succès de Flash qui ne fait, malgré tout, pas l'unanimité. En effet, on repproche rapidement au langage d'être difficile à appréhender. L'exécution d'un fichier flash réclame également le téléchargement et l'installation d'un logiciel tiers (lecteur) sur le navigateur du client. Mais Adobe défend son produit et les téléchargements et l'installation sont facilités au point qu'aujourd'hui le lecteur flash est le logiciel le plus téléchargé et installé au monde. Il touche (toujours d'après Adobe [3]) plus de 99% du parc informatique. Contrairement à Javascript, la portabilité et l'homogénéité d'exécution des fichiers flash est garantie par l'utilisation obligatoire de son lecteur atitré maintenu par Adobe et disponible actuellement pour tous les navigateurs.

Ces langages permettant une adaptation dynamique de l'interface chez le client vont modifier progressivement la manière dont on aborde le client web. Au départ, ces langages seront incorporés au compte goutte dans les interfaces pour apporter une touche dynamique ponctuelle. Ces langages dynamiques vont par la suite occuper une place plus prépondérante et le client va devoir être conçu comme une application à part entière qui interragit avec un serveur distant. Le modèle utilisateur/serveur devient un modèle distribué où le client est une "application" web. Le client passif des débuts (simple affichage d'HTML) cède la place à un client acteur de son interface. Il devient alors un client "riche" tandis que le serveur

devient de plus en plus un fournisseur de données plutôt qu'un fournisseur de pages.

4.1.2 Choix d'un client riche

Au vue des attentes énoncées dans l'analyse des exigences, on cerne vite qu'une des difficultés essentielles de l'interface est de présenter des données complexes et leurs relations avec une certaine fluidité. Il est donc inenvisageable de se contenter d'une interface statique du web classique à l'ancienne. Au contraire, l'interface devra permettre de déployer les résultats de recherche et ce indéfiniment si nécessaire. L'interface devra être dynamique et autant autonome que possible du serveur afin d'éviter les ralentis liés à une communication réseau trop importante. On se situe clairement dans le cas de figure d'un client riche.

Zope , à l'époque, proposait déjà un fonctionnement plus dynamique que le couplage PHP/XHTML statique de base. Il présentait en revanche trois défauts majeurs mis en évidence dans la première partie de ce mémoire et que nous allons tenter d'éviter en choisissant notre technologie de développement.

- Le développement en Zope se fait automatiquement dans l'univers de programmation prévu à cet effet au sein d'un navigateur. Bien que ça ne présente pas un inconvénient majeur, cet univers de programmation n'est pas flexible et, à l'image du reste du framework, l'univers de développement, lui aussi, n'évolue plus.
- Certaines fonctionnalités en Javascript utilisées par le framework sont incompatibles avec Internet Explorer ce qui pose des problèmes de portabilité. Cette difficulté présente un frein majeur dans le domaine où l'application devrait être déployée et devra donc être prise en compte lors du choix de technologie.
- Zope a été utilisé largement mais au travers de certains frameworks spécifiques dont Plone en particulier. La plupart de la communauté utilisant Zope 2 dans le contexte de ce framework annexe, Plone a profité de nombreuses améliorations dont n'a pas profité Zope. Cette utilisation canalisée a fait de Plone un outil majeur au détriment de son parent Zope qui, à force de ne pas évoluer, a fini par être abandonné. Si c'est le risque obligatoire pris par n'importe quel programmeur face à un choix technologique, il peut être minimisé en prenant des technologies plus éprouvées, plus répandues, avec des communautés d'utilisateurs diversifiées qui en garantissent la pérénité.

Afin d'éviter les désagréments rencontrés avec Zope, nous nous intéresserons à des technologies éprouvées et utilisées largement à l'heure actuelle. Dans le développement d'applications clientes riches, il existe des solutions libres, propriétaires, et même des solutions intermédiaires [4]. Si les solutions propriétaires (Microsoft .NET par exemple) offrent certaines garanties liées à l'entreprise qui les produit (une bonne évolutivité, maintenance,...), elles posent une difficulté financière qu'actuellement le projet Expesurf ne peut adresser. Dans l'univers libre la solution la plus répandue dans le développement d'applications riches est l'AJAX (Asynchronous Javascript and XML). Ce marché est concurrencé actuellement par Flex, une solution semi-libre proposée par Adobe. Il existe d'autres solutions puissantes également, telles que JavaFx [7] par exemple, mais qui sont en pleine conquête de leur marché actuellement et n'offrent donc pas la communauté et les garanties d'une solution éprouvée et fort répandue.

L'univers d'AJAX est un petit peu particulier puisqu'il ne propose pas "une" technologie spécifique mais une méthode d'utilisation d'un ensemble de technologies existantes : HTML, XML, l'objet XMLHttpRequest, le CSS et le Javascript. L'objet XMLHttpRequest, la seule technologie à n'avoir pas encore été abordée propose un cadre d'échange des données entre le client et le serveur. La méthodologie AJAX est facilitée au moyen de différents frameworks tels que Dojo, Google web toolkit, ect. Une caractéristique majeure qui avantage l'utilisation d'AJAX par rapport à ses concurrents est le fait qu'il utilise des technologies existantes et bien connues des programmeurs actuels. Seule la méthodologie doit donc être apréhendée et l'apprentissage d'AJAX est donc plus rapide pour une équipe expérimentée dans le développement web. AJAX permet également à des sites web existants développés en HTML "classique" de se moderniser et de les rendre dynamiques sans refondre complètement l'architecture du site mais simplement en incorporant et en adaptant le site avec les autres langages. AJAX est également parfaitement portable. Il utilise des librairies Javascript mais les librairies sont choisies de manières à adapter leur comportement d'un navigateur à l'autre. L'inconvénient de cette pratique est que la portabilité de l'application dépend toujours de la portabilité de ses librairies et donc de leur maintien. Cette dépendance implique une prise de risque au niveau du développement lui-même en plus de la portabilité : au même titre que pour l'adaptateur à la base de données PSQL de Zope, qu'arrive-t-il le jour où une librairie-clé n'est plus maintenue ? En revanche, AJAX, et plus particulièrement son framework Dojo ont

été utilisés pour le développement de l'interface d'acquisition de connaissances d'Expesurf. Ceci peut être perçu comme un avantage de minimiser la diversité des technologies employées dans ce programme qui en compte déjà quelques unes. A chaque nouvelle technologie employée, la maintenance du logiciel exige une expertise dans un domaine supplémentaire. De plus si à terme les deux interfaces devaient communiquer ou être fusionnées il serait intéressant qu'elles partagent une même technologie.

Flex présente les avantages et inconvénients d'un développement en flash. En effet, Flex propose une nouvelle technologie à part entière plutôt qu'un assemblage de l'existant. Afin d'accompagner le programmeur dans ses développements flash, Adobe propose un nouveau langage : le MXML. Il vise à faciliter le développement en flash afin de permettre le développement d'applications à part entière plutôt qu'à incorporer de simples animations au sein de sites développés en HTML/PHP. On abandonne, avec Flex, définitivement l'HTML comme langage de présentation pour un fichier flash. Il est obtenu en compilant un mélange de MXML (description statique des interfaces dans un type d'XML) avec de l'Actionscript utilisé pour décrire la dynamique de l'application : les opérations à effectuer lors d'un click, les lectures de vidéo, les manipulations de variables, de données, les appels au serveur etc (voir figure 4.2). En réalité la description d'interfaces en Actionscript est plus ardue. Le MXML va donc être interprété pour produire le code Actionscript correspondant. Le code produit (de l'Actionscript pur) sera alors lui aussi interprété pour produire du flash. Le fichier flash sera envoyé chez le client (voir 4.3). Ce dernier fichier sera compilé sur son terminal au moyen du flashplayer qui produira un fichier directement exécutable. Afin de respecter certaines normes d'ergonomie et d'accessibilité les feuilles de style CSS seront progressivement intégrées dans l'environnement Flex pour y occuper un rôle important dans la dernière version. Si Flex résoud la problématique du développement du client, il laisse une marge de décisions quant à la manière dont on peut aborder la communication client/serveur, et comment on peut développer la partie serveur.

Il est intéressant de noter que Flex est semi-libre. Sur ce plan, AJAX et Flex divergent quelque peu. Ajax est complètement libre, en tout cas de nombreux framework en faisant usage le sont. Flex propose une licence plus nuancée [10]. Tout le monde peut ouvrir et éditer des fichiers flash et flex avec n'importe quel éditeur/outil de développement. De même, le Flex SDK est open source et gratuit. La nuance se situe au niveau du flashplayer : disponible gratuitement mais sous licence d'Adobe. Cela présente un avantage : Adobe garde la main mise sur sa technologie et garantit donc une uniformité d'exécution que n'a pas rencontré Javascript. En revanche, ce qui n'est absolument pas libre, c'est l'outil de développement FlashBuilder. Cet IDE (Integrated development environment) construit en Java sur base de l'IDE Eclipse peut être téléchargé en tant qu'application native ou en tant que plugin sous Eclipse. Adobe offre actuellement une licence gratuite pour tout enseignant ou étudiant mais est très cher dans tout autre contexte. S'il ne constitue pas l'unique manière de développer en Flex actuellement, il reste néanmoins l'outil le plus avancé. Des alternatives open source complètes existent (FlexBeans, plugin sous NetBeans [1], FlashDevelop,...). Elles présentent deux inconvénients : des communautés d'utilisateurs moins vastes et moins organisées et ensuite, moins de fonctionnalités notamment dans l'approche WYSIWYG des interfaces en MXML (moins puissante que sous FlashBuilder).

Une étude comparative de *Forrester* [14] [2] dresse un bilan des avantages et inconvénients du choix technologique de Flex par rapport à AJAX. Elle met en évidence la différence de courbe d'apprentissage de Flex, plus difficile à appréhender pour un développeur averti en WEB, à l'inverse d'AJAX. En revanche, l'apparente portabilité d'AJAX était mise à mal par le fait qu'elle resterait toujours une solution sur mesure pour le navigateur contrairement à Flex qui propose une réelle portabilité. Forrester estime qu'Ajax rencontre tous ses avantages dans la modernisation d'anciennes architectures web mais que, pour de nouvelles implémentations, il est préférable de choisir Flex auquel il projete un avenir plus prometteur [6].

Ni l'un ni l'autre des choix ne remporte un avantage absolu (voir tableau 4.1.2). En revanche, Flex semble offrir de plus grandes possibilités de développement. L'autonomie plus importante du client par rapport au serveur et une exécution indépendante de l'application permettent d'utiliser davantage de ressources. La communication avec le serveur est optimisée par rapport aux possibilités qu'offre AJAX. Malgré la bonne portabilité des deux types de développement, la portabilité des développements en Flex dépendent d'un seul acteur tandis que la portabilité d'un développement en AJAX exige une bonne maintenance de toutes les librairies dont dépendent l'application. Puisque nous ne redynamisons pas une

[1] IDE concurrent d'eclipse développé par sun

[2] Forrester research est une entreprise indépendante effectuant des recherches dans le milieu de la technologie et de ses marchés

Critère	**AJAX**	**Flex**
Intégration au système existant	Une fusion du développement avec l'interface existante sera plus aisée	Une intégration partielle est possible mais les deux interfaces resteront distinctes
Outils de développement	Nombreux outils de développements gratuits et puissants	FlashBuilder payant [3] et autres outils complets gratuits
Accessibilité des sources	Sources accessibles avec n'importe quel éditeur de texte	Sources également accessibles avec n'importe quel éditeur
Apprentissage du langage	Pour un développeur web expérimenté, plus aisé qu'en Flex	Longue courbe d'apprentissage
Ouverture du langage	Les frameworks permettant de développer en AJAX varient entre des solutions entièrement open source et des frameworksp propriétaires payants.	Flex et ActionScript sont open source. En revanche, le FlashPlayer ne l'est pas.
Portabilité	Bonne mais dépendante de la librairie et du framework	Bonne mais dépendante du maintien de FlashPlayer par Adobe
Evolution	Dépend du choix du choix du framework et du maintien des librairies	Dépend du suivi par Adobe
Soutien	Communauté vaste et active	Communauté vaste et active

TAB. 4.1 – Tableau comparatif AJAX et Flex

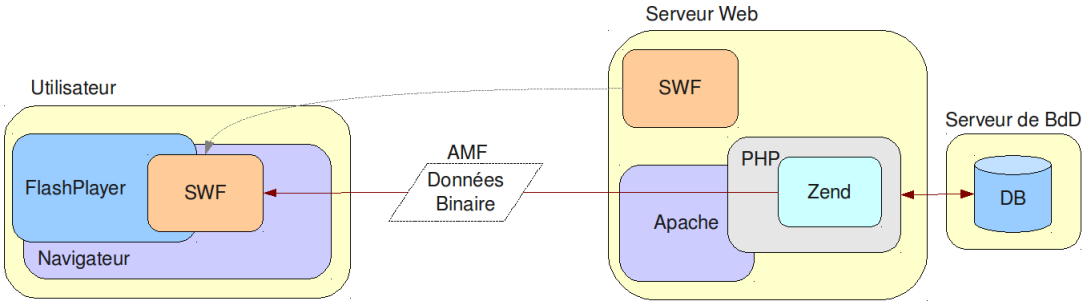


FIG. 4.3 – Architecture d'une application en Flex

interface existante et que la portabilité de l'application est primordiale, Flex semble mieux adapter au contexte de notre développement.

4.1.3 Couche Middleware

Comme mentionné plus haut, Flex propose une solution spécifique pour le développement du client riche mais laisse quelques libertés quant à la technologie employée pour les échanges client/serveur. Il s'agit finalement de choisir un formalisme de communication entre le client et le serveur. Il existe différentes solutions (HTTP Services, services PHP, web services,...) prises en charge par FlashBuilder. Différentes études donnent des résultats d'éfficacité contradictoires en fonction du protocol d'échange choisi et des types de données circulant. En définitive, les écarts d'efficacité sont négligeables d'une méthode à l'autre surtout si l'on considère que la transmission de données n'est peut-être pas l'unique point de perte d'efficacité possible. Les phases de traitement préalable des données auprès du serveur et le temps de présentation des données du côté du client peuvent s'avérer plus gourmandes encore que le temps de transmission. Une des technologies proposées par le framework Zend est préconisée par Adobe et fait l'objet d'un partenariat privilégié. Le fait qu'Adobe se porte garant de cette solution assure une certaine pérénité à celle-ci qui, dans le cadre de ce projet, l'avantage. C'est pour cette raison que nous avons arrêté notre choix sur celle-ci. Le framework Zend se comporte en véritable middleware. A la compilation Zend génère automatiquement, sur base de classes de services PHP sur le serveur (développées par le programmeur), les couches intermédiaires qui vont permettre des appels distants de procédures depuis le client vers le serveur. Les données circulant utilisent dans ce cas le protocole AMF (Action Message Format) [13] qui est un format de fichiers binaires représentant des objets Actionscript séréalisés puis compressés. Ce type de données est utilisé au sein du FlashPlayer pour le transfert et la sauvegarde des données.

4.1.4 Côté serveur

Les développements chez le serveur présentaient moins de choix technologiques décisifs bien que deux décisions intéressantes aient été prises. La première visait à assurer la pérénité de l'application également. En effet, à partir de PHP6, l'utilisation de PDO (PHP Data Objects) afin de communiquer avec la base de données. La librairie PDO construit une couche d'abstraction par dessus les manipulations proposées par les différents pilotes bases de données. Il permet ainsi, en modifiant uniquement le string de connection de changer de base de données plutôt que de modifier l'intégralité des accès à la base [16]. PDO propose des méthodes permettant de se protéger contre les injections SQL sans passer par des requêtes préparées et de toutes sortes d'autres petites subtilités spécifiques au langage PHP. Si on peut douter de l'intérêt majeur d'avoir la possibilité de changer de gestionnaire de bases de données au complet, ce type d'opération étant plus rare, l'utilisation de PDO deviendrait obligatoire à partir de PHP 6 (la configuration actuelle étant en PHP 5). Tous les accès à la base de données sont donc conformes à la prochaine version du langage.

Le deuxième choix technique impliquait de déterminer comment on allait gérer les sessions d'utilisateurs. Flash arrivant non compilé chez le client (fichier .swf), il est facile pour un utilisateur averti de s'approprier l'implémentation et de la détourner de ses objectifs initiaux. Si les données de session sont strictement comprises et vérifiées dans l'interface en Flash, il est facile d'ouvrir le fichier Flash et d'éditer les sections limitant ses privilèges pour s'en attribuer d'autres. De ce fait, l'authentification et les

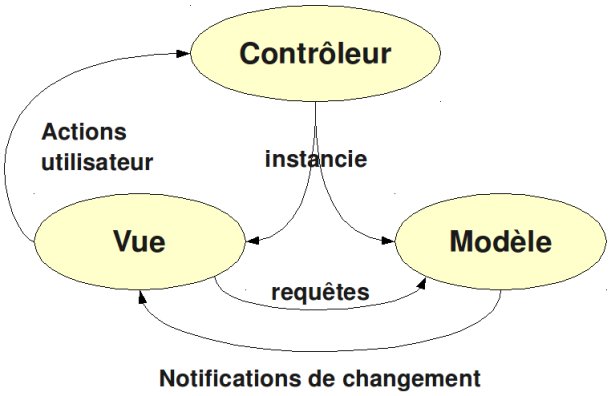


FIG. 4.4 – Le modèle MVC au sein de l'interface AC

vérifications de droits dans les opérations critiques doivent être effectuées en PHP. Zope proposait une gestion des sessions d'utilisateurs qui déservait l'interface d'acquisition de connaissances mais également l'interface d'acquisition de données. Se passer de Zope définitivement, à terme, implique de remplacer ce système de session d'utilisateurs. De ce fait, afin de limiter les modifications sur la deuxième interface, il serait intéressant de proposer un système de session similaire, proposé dans un module isolé de manière à ce qu'il puisse être réadapté par la suite pour répondre aux attentes des deux interfaces sans réclamer de changements majeurs chez l'une et l'autre. En l'occurrence, le système de session proposé par Zope modifie les données de session retenues par le navigateur de l'utilisateur et accessibles telles des variables globales en tout point du site. C'est donc ce type de données qui seront modifiées par le module session en PHP dédié à cet effet.

4.1.5 Le modèle MVC

La technique MVC a été décrite pour la première fois en 1979 par Trygve Reenskaug. [12] [11] L'objectif de son travail visait à définir une architecture type qui permettrait de présenter de différentes manières une grande quantité de données. Depuis lors, son architecture a suscité un intérêt très important et ses principes sont devenus des méthodes de programmation incontournables dans les développements d'interfaces homme/machine.

Il s'agit d'une méthode de conception définissant les bases de l'architecture de l'interface d'une application. Il la divise en trois composants : le modèle, la vue et le contrôleur. Le modèle se présente comme un gestionnaire et détenteur des informations. Il les puise, les altère et annonce les changements qu'elles encourent. Une vue est un mode de représentation des informations. Elle agit comme un filtre visuel pour le modèle auquel elle est associée. Si un modèle peut être représenté par plusieurs vues, une vue en revanche est toujours associée à un seul modèle. Le contrôleur va permettre la coordination entre ces éléments. Il constitue une interface entre l'utilisateur et la vue. Il va déterminer ce que voit l'utilisateur et interpréter les demandes de l'utilisateur afin de modifier la vue en fonction de celles-ci.

Cette description de base suffit à la compréhension globale du rôle des trois composants, mais elle est sujette à interprétation. Et d'ailleurs, elle sera interprétée pour produire de nombreuses variantes du modèle définissant quel composant contacte quel autre et comment. De ce fait, il est intéressant de définir le modèle MVC tel qu'il a été défini dans la construction de notre application et de comprendre ensuite comment ce modèle peut être implémenté en Flex.

L'architecture implémentée au sein de l'interface d'acquisition de connaissances se présente comme à la figure 4.4. La structure reprend les grands composants de l'architecture MVC. Ici le modèle crée (instancie) les différents modèles et les vues qui en dépendent. Les vues sont prévenues par les modèles (de manière passive) lorsqu'ils sont sujets à des modifications. Les modèles de leur côté chargent les informations, les préparent, les modifient, et répondent aux requêtes des vues. Les vues sont instanciées par le contrôleur. Elles lui transmettent les signaux qu'elles reçoivent de l'utilisateur pour qu'il avise de la réaction à avoir.

Si ce modèle est aussi apprécié, c'est parce qu'il permet de modifier très simplement le mode de représentation des informations parce que tout ce qui les compose est isolé du reste de l'application. De

la même manière, s'il est modifié, le mode d'accès à la base de données ne le sera qu'une fois, dans le modèle, pour toutes les vues qui en dépendent. De plus, l'association d'un modèle à autant de vues que l'on veut permet de centraliser des traitements semblables et de s'assurer d'une meilleure modularité de l'application.

C'est donc en gardant à l'esprit cette décomposition du code source des interfaces homme/machine, que les nouveaux framework se développent. Ils permettent ainsi de développer plus rapidement et plus simplement des modèles MVC. En Flex, une partie de la distinction entre les trois composants est facilitée ainsi qu'expliqué par la suite. Mais d'autres framework ont été développés par dessus Flex et qui permettent un développement selon une version du modèle et efficacement.

4.1.6 Présentation du framework Flex

Flex est depuis Flex 3[4] un framework open source. Présenté depuis 2003 et développé par Adobe depuis 2006, Flex permet le développement de l'application cliente dans le cadre d'une architecture client riche/serveur notamment au travers de la technologie Flash. A cet effet, elle utilise deux langages de programmation : ActionScript 3 et MXML. Les deux langages ont des spécificités particulières. Ils interagissent entre eux ainsi qu'avec le serveur selon des modalités que nous allons présenter.

4.1.6.1 L'ActionScript 3 et l'utilisation du framework Zend

Le langage ActionScript 3 est un langage orienté objet qui supporte le travail procédural des applications développées en Flex. Il peut servir à construire des composants de l'interface également mais cette tâche est largement facilitée au travers de l'utilisation du langage MXML, plus intuitif, que l'on décrira plus longuement par la suite.

De ce fait, au sein d'une architecture construite sur le modèle MVC, seront développés en ActionScript le modèle et le contrôleur. Toute la communication du client vers le serveur passera par des classes en ActionScript. Cette communication est schématisée à la figure 4.5.

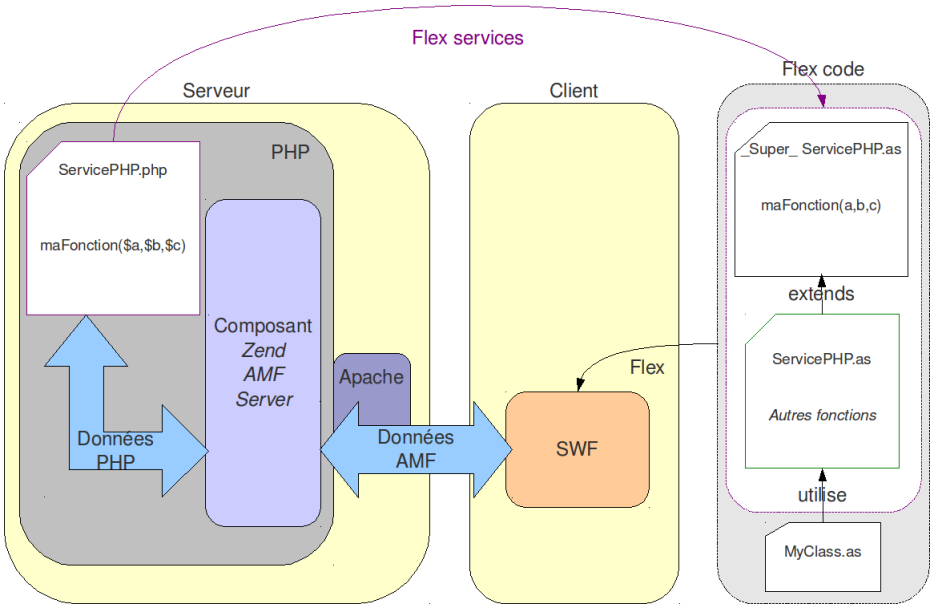


FIG. 4.5 – Echanges client serveur avec Zend AMF

Sur le serveur, les échanges d'informations entre le FlashPlayer et PHP sont rendus possibles par l'intermédiaire du composant *AMF Server* du framework Zend [1]. Le composant Zend AMF va publier les services décrits dans un ensemble de classes écrites en PHP[5]

[4]La version actuelle est Flex 4.

[5]Dans l'exemple du schéma, nous avons utilisé la classe *ServicePHP*. Elle comporte une méthode : «maFonction($a,$b,$c)» en PHP également.

Lors du développement de notre application, il faudra indiquer à Flex quels sont les services proposés et de quel type ils sont. En l'occurrence, il s'agit de services de type Zend AMF. On couplera à chaque classe de services en PHP, une classe écrite en ActionScript qui va masquer l'appel distant[6] au serveur. Dans le cadre de notre développement, nous avons recours au module *Flex Services* de FlashBuilder (outil de développement décrit par la suite) et qui va configurer les fichiers déclarant les services et produire les classes ActionScript homologues aux services PHP[7]. La flèche en mauve (sur le schéma 4.5) indique la génération de la classe homologue (`_Super_ServicePHP.as`) à partir de la classe de services (`ServicePHP.php`). Elle comprend la méthode «maFonction(a,b,c)» qui masque l'appel distant à la méthode «maFonction($a,$b,$c)» sur le serveur.

Chaque changement significatif des services proposés par la classe `ServicePHP.php` (nouvelle méthode, changement d'arguments, ...), de vra être répercuté sur sa classe homologue en ActionScript. Lors d'une mise à jour d'un service avec l'outil *Flex services*, la classe homologue est entièrement regénérée. Afin de permettre au programmeur de compléter cet objet d'autres fonctions sans qu'elles ne soient supprimée à chaque mise à jour, une autre classe (`ServicePHP.as`), qui étend la précédente, est créée. Cette classe sera celle employée dans le code de du composant client pour effectuer des appels distants.

Après compilation, le fichier Flash résultant est exécuté chez le client. Il utilise le format AMF pour communiquer avec le serveur. Les appels sont transmis au composant *Zend AMF Server* qui transforme les paramètres encodés au format AMF (qui sont en fait des objets ActionScript séréalisés) en objets PHP équivalents[8]. Il identifie ensuite la classe PHP qui peut exécuter cet appel avec les nouveaux arguments. La valeur de retour de la fonction en PHP est à son tour traduite en objet ActionScript, séréalisée, et transmise au client.

Il existe un ensemble de correspondances entre les types d'objets PHP et les types d'objets ActionScript. C'est cet ensemble de correspondances qui va dicter le type des paramètres et de la valeur de retour d'un appel distant. Dans le cadre de ce développement, les échanges entre le serveur et le client comprendront des types primitifs, qui se traduisent par leur homologue évident d'un langage à l'autre. Nous échangerons aussi régulièrement des tableaux à deux dimensions représentant le résultat d'une requête : un extrait de table ou de vue. Ce type de données plus complexe est en php un simple *array*. En ActionScript il prend la forme d'une *ArrayCollection*. Une ArrayCollection est un tableau organisé pouvant être utilisé avec un accès indexé aux données mais également, comme en PHP, un accès par référence. Le type fait l'objet d'une attention particulière au sein du langage puisqu'il présente la première source de données de toutes les applications. De ce fait il est lié à de nombreux types de composants visuels tel qu'on le verra par la suite.

Ainsi, dans une classe ActionScript ailleurs dans le programme, on pourra importer puis créer des objets de type *ServicePHP*[9] et appeller sur lui les méthodes distantes comme si on les appellait localement ou presque ... En effet, les appels distants au travers de Zend AMF sont toujours asynchrones. Cela signifie qu'après l'appel de la fonction l'application n'attend pas la valeur de retour de l'appel pour continuer son traitement. Par exemple, si nous prenons le code à la table 4.1.6.1, nous importons notre classe ServicePHP à la ligne 2. Nous créons également un objet de type particulier : *CallResponder* (ligne 5). En effet, nous ne recevons pas le résultat directement lorsque nous appellons la fonction distante puisqu'il s'agit d'un appel asynchrone. Un objet de type CallResponder est spécialement conçu pour surveiller l'appel. C'est son attribut *token* qui le lie à un appel de méthode précis (ligne 10). En revanche, s'il guète la valeur de retour le CallReponser n'en fait rien.

Afin d'être averti de la réception de la réponse, et d'adopter un comportement adapté, on associera à notre *CallResponder* un autre objet appellé action listener . (voir ligne 11). Un action listener est défini au travers d'un événement et d'une procédure. A l'occurrence de l'événement, l'action listener appelle la procédure. Il peut être associé à différents types d'évênements et en particulier à l'évênement déclenché par l'arrivée d'un résultat depuis le serveur. Du fait de cet appel asynchrone et des classes générées en ActionScript on retrouvera toujours la même structure d'une classe utilisant un ensemble de services opèrera toujours en deux temps :

1. effectuer l'appel distant et lui associer un action listener à la réception du résultat

[6] On appelle *appel distant* un appel de méthode impliquant une communication réseau. C'est un appel qui ne s'exécute pas au sein du composant qui le sollicite.

[7] La configuration et la production des classes n'est pas complexe en soi mais répétitive.

[8] Il existe des règles de conversions précises d'objets ActionScript vers des objets homologues PHP.

[9] Par convention dans le développement de l'interface toutes les classes de services sont accompagnées par le suffixe «PHP». De cette manière, lors de l'utilisation des classes générées au sein du reste du code ActionScript, on lit mieux qu'il s'agit d'un appel à une classe PHP.

```
1       ... //autres importations utiles
2      import services.ServicePHP;
3
4      public var ServicePHP serv = new ServicePHP();
5      public var CallResponder servResponder = new CallResponder();
6
7      public function action():void
8     {
9           servResponder.token = serv.maFonction(x,y,z);
10          serv.Responder.addEventListener(ResultEvent.RESULT,reaction);
11          //je continue mes traitements
12    }
13
14    public function reaction(event:ResultEvent):void{
15          var Object obj = event.result;
16          //utiliser le résultat obj
17    }
18
19     ...
```

TAB. 4.2 – Exemple de code ActionScript effectuant un appel à un serveur dans le corps d'une classe quelconque

2. le résultat est réceptionné, l'action listener associé enclenche l'exécution d'une méthode manipulant le résultat obtenu

L'ensemble des classes implémentant ces appels en deux temps rapatrient les données du serveur et constituent essentiellement les Modèles de notre architecture MVC.

Le contrôleur de l'application est plus diffus. En effet, dans le développement d'une interface en particulier, les événements liés à la présentation entraînent les réactions du contrôleur. Or, les évènements liés à la présentation sont définis dans la couche de présentation. Les deux sont donc plus difficilement dissociables que dans le reste de l'application. A ce titre, la distinction MXML/ActionScript proposée par Flex facilite considérablement la séparation de la vue et du contrôleur. En effet, au sein d'un fichier, les développements en ActionScript constituent la composante contrôleur associée à la vue décrite par le MXML. Le couple Contrôleur/Vue sera donc présent dans tous les fichiers MXML et en particulier le fichier principal de l'application (Control.mxml) est un fichier MXML.

4.1.6.2 Présentation en Flex, le langage MXML

Si la couche de présentation peut être construite entièrement en ActionScript, ce langage n'a pas été conçu pour ça et par la suite, le travail s'avère long et fastidieux. C'est toute l'ingéniosité de Flex que d'avoir proposé le MXML. Il s'agit d'un langage XML spécialement adapté à la construction d'interfaces. Deux grandes librairies existent et permettent de composer une interface : une librairie de composants initiaux (MX) et une librairie plus récente (Spark), préférée par Flex puisqu'elle apporte un nouvel accent sur la compatibilité des composants avec des normes d'accessibilité et une utilisation plus rigoureuse du CSS pour définir l'allure de l'interface. Malheureusement, cette dernière comporte encore quelques manquements et, comme tous les composants proposés par MX sont encore maintenus, il arrive d'y recourir pour certaines fonctionnalités. Les fichiers MXML commencent avec l'en-tête suivante de la table 4.1.6.2. Elle présente la version de l'XML (ligne 1), son encodage et les inclusions de librairies avec leur chemin d'accès (ligne 2).

Les composants MXML peuvent être de natures très différentes. Vu leur utilisation privilégiée dans la présentation de l'application, ils vont bien entendu définir des éléments de l'interface. Un composant MXML peut être un bouton, un panel ou une grille de résultats. Mais les composants MXML ne sont pas toujours des éléments visuels. On peut par exemple déclarer des variables : tableaux, chaînes de caractères, action listener, etc. Tous les éléments ActionScript ne sont pas traduisibles en MXML mais, à l'inverse, tout composant MXML peut être traduit en ActionScript. D'ailleurs, c'est la transformation qu'effectue Flex. La conception d'une interface au travers des balises MXML est très intuitive. On va

```
1       <?xml version="1.0" encoding="utf-8"?>
2       <s:Application xmlns:fx="..." xmlns:s="..." xmlns:mx="...">
3
4           ...
5
6       </s:Application>
```

TAB. 4.3 –

déclarer vouloir une fenêtre, au sein de cette fenêtre on va placer un bouton, une image, un texte. Ces éléments de présentation sont dotés de propriétés définies sous la forme de tags au sein de la balise. Ces méta-données peuvent être de différentes natures. Elles peuvent contenir des informations procédurales et des informations statiques. Par exemple (table 4.1.6.2), un bouton est doté d'un *label* qui indique l'intitulé du bouton (ligne 1). Il peut être doté d'un identifiant qui permettra de le référencer au sein du code ActionScript par la suite.

```
1       <s:Button label="Test" id="zeBouton"/>
```

TAB. 4.4 –

En revanche, certaines caractéristiques du bouton sont dynamiques. Elles réclament l'appel d'une procédure qui va définir l'action du bouton. Par exemple, lors d'un «click», c'est au travers de ce type de meta données que le code de présentation de l'interface en MXML va exécuter le code procédural en ActionScript. Imaginons une interface très simple, toujours avec notre bouton, mais également un label défini à la ligne 2 du code 4.1.6.2 (zone de texte sur une ligne). Lorsque l'on clique sur le bouton, nous souhaitons que le label indique "Bonjour!". Nous définissons donc un label sans préciser le texte qu'il doit afficher (accessible au travers de la méta donnée *text*) et complétons le code à exécuter en cas de click sur notre bouton.

```
1       <s:Label id="zeLabel"/>
2       <s:Button label="Test" id="zeBouton" click="zeLabel.text='Bonjour!';"/>
```

TAB. 4.5 –

Remarquez que le code ActionScript qui suit click accède, au travers de son identifiant du label (*zeLabel*) aux méta données du bouton (ligne 3). Il peut alors leur assigner des valeurs. Dans cet extrait de code on peut effectuer de nombreuses manipulations comprenant des appels de procédures. Toute l'étendue de la puissance du langage ActionScript est accessible au travers de ces méta données. De ce fait, il est intéressant de coupler l'élément de présentation de notre document MXML avec un extrait de code source où il va pouvoir enregistrer des données, les manipuler, et gérer les réactions aux opérations de l'utilisateur (code de la table 4.1.6.2). La balise *script* permet d'isoler du code ActionScript. Les zones encadrées par les balises CDATA ne sont pas parsées en tant que MXML ce qui permet d'utiliser les caractères autrement protégés tels que les opérateurs >, <, etc.

Nous pouvons donc étendre le code actuel afin que l'objet du click soit en fait une procédure à part entière comme dans la table 4.1.6.2, ligne 3.

Afin de garder le code lisible et de continuer à bien distinguer la présentation du reste de l'application, il est important lorsque les procédures deviennent plus complexes de les définir dans une zone de script. Cette zone de script va nous permettre également de définir des variables. Rien que le petit extrait de code construit jusqu'ici tranche déjà avec un script classique de web. Afin de présenter le "Bonjour", il aurait fallu que la page contenant le bouton soit rechargée et que le serveur renvoie une page contenant l'indication textuelle "Bonjour!". A l'inverse, en dehors des appels à des informations de la base de données, tout le traitement s'effectue sur la machine du client. De ce fait, des interfaces plus interactives qui réclament de nombreuses petites modifications doivent faire intervenir une technologie tierce pour se charger des parties dynamiques de l'interface.

```
1   <f:script>
2       <![CDATA[
3           //Définir ici toutes les opérations ActionScript
4       ]]>
5   </f:script>
```

TAB. 4.6 –

```
1   <f:script>
2       <![CDATA[
3           public function button_click():void{
4               zeLabel.text = "Bonjour!";
5           }
6       ]]>
7   </f:script>
8
9   <s:Label id="zeLabel"/>
10  <s:Button label="Test" id="zeBouton" click="button_click()"/>
```

TAB. 4.7 –

Le dynamisme de notre petit exemple reste tout de même terriblement basique et il présente la difficulté, que malgré que la zone de texte a été altérée, elle n'en restait pas moins déclarée en mxml. Il peut arriver que l'on veuille faire apparaître des composants au sens strict et qu'ils ne doivent pas occuper la présentation avant d'être sollicités. Les apparitions et disparitions de composants peuvent être gérées au travers d'ActionScript également. Pour cela, il nous faudra un composant de type conteneur que l'on va pouvoir populer d'autres composants fils. Ces conteneurs peuvent être des onglets, des panels, ... ActionScript définit en fait un tableau comprenant les enfants d'un conteneur. A ce tableau on va pouvoir ajouter et retirer des éléments. La librairie Spark ne contient pas encore de conteneur capable de jouer ce rôle. Nous utiliserons une VBox de type mx (table 4.1.6.2 ligne 1). Le composant VBox est une boîte dont les enfants seront alignés verticalement. Nous allons placer dans la boite parente un label et un bouton de nouveau mais également un champ de saisie de texte. Le tag "text" précisera l'indication qui précède le champ.

```
1   <mx:VBox id="boiteParente">
2       <s:Label id="zeLabel"/>
3       <s:TextInput id="saisieNom" text="Utilisateur :"/>
4       <s:Button label="Test" id="zeBouton" click="button_click()"/>
5   </mx:VBox>
```

TAB. 4.8 –

La modification intéressante est au niveau de l'ActionScript. Nous considérons un ensemble d'utilisateurs connus. Lorsque je complète mon nom dans le champ de saisie, et que je clique sur le bouton, le script va vérifier s'il connait l'utilisateur (au travers de la fonction de la table 4.1.6.2 à partir de la ligne 3). Si oui, l'utilisateur sera salué (ligne 8 ou 12) et sa photo apparaîtra en haut de la page (ligne 7 ou 11). Sinon, il indiquera qu'il ne connait pas l'utilisateur (ligne 14).

Le langage MXML définit également une autre fonctionnalité, très puissante : *Data binding*. D'ailleurs en règle générale, l'ensemble des fonctionnalités proposées par MXML en guise de remplacement de fonctionnalités similaires en Javascript ne sont jamais superflues. Leur développement en ActionScript seul serait long et fastidieux. Le data binding consiste à lier un élément visuel à une source de données. Une modification de la source de donnée implique alors une modification de la vue à laquelle elle est liée. Tous les types de données ne peuvent pas être source de data binding mais les types simples le sont et certains types plus complexes tels que l'ArrayCollection également. Dans notre exemple, nous assignons systématiquement une valeur au champ textuel de zeLabel. Nous pourrions lier zeLabel à une variable, par exemple la variable *nom*). Si nous modifierions la valeur de la variable, le champ serait ainsi

```
1     <f:script>
2         <![CDATA[
3             public function button_click():void{
4                 var img:Image = new Image;
5                 if(saisieNom.text=="Sandy"){
6                     img.source="sandy.jpg";
7                     boiteParente.addChildAt(img,0);
8                     zeLabel.text = "Bienvenue Sandy";
9                 } elseif(saisieNom.text=="Mathieu"){
10                    img.source="mathieu.jpg";
11                    boiteParente.addChildAt(img,0);
12                    zeLabel.text = "Bienvenue Mathieu";
13                } else {
14                    zeLabel.text = "Je ne vous connais pas";
15                }
16            }
17        ]]>
18    </f:script>
```

TAB. 4.9 –

automatiquement modifié. Afin d'indiquer à ActionScript que les vues liées à la donnée doivent être mises à jour, la donnée doit être déclarée comme donnée *[Bindable]*. De même, on indique par des acolades le fait que l'information est passée en argument et non à prendre comme une constante. Sur notre exemple cela donnerait le code de la table 4.1.6.2.[10]

Remarquons que l'on peut se permettre entre ces accolades (ligne 25) d'inclure, encore une fois, du code ActionScript. Il devra renvoyer des données du type attendu par le tag du composant. Ce faisant il nous économise ici l'écriture de "Bienvenue" pour chaque utilisateur. Évidemment dans notre exemple ça ne présente pas d'intérêt majeur mais pour de plus grandes quantités de données, ou lorsque différentes vues commencent à être associées à la donnée il devient très intéressant de pouvoir rafraîchir toutes les vues sans réassigner la variable. De même dans l'architecture MVC que nous avons décrite précédemment, une modification du modèle se répercute directement sur les vues dont elle dépend.

Notons dans le dernier exemple que les composants visuels sont pour la plupart redimensionables. On peut définir qu'ils ont une largeur fixe ou proportionnelle à l'espace dont ils disposent définis par la taille du composant parent. Dans ce cas on utilise des valeurs en pourcentage comme c'est le cas de notre boite parente qui occupera 90% en largeur et en hauteur de l'espace que lui réserve son contenant. Sans ces informations certains composants épousent la taille de l'ensemble de leur contenu ; d'autres, ont un format par défaut.

Le langage MXML, au delà du fait qu'il propose une couche par dessus ActionScript pour manipuler les interface de manière plus aisée, est réellement puissant. Il comporte un grand nombre de composants fort diversifiés comprenant des effets visuels, des composants multimédia, des éléments de disposition définissant des mises en pages flexibles etc. Ensuite, MXML propose pour chacun de ces composants de nombreux tags définissant des fonctionnalités qui simplifient considérablement le travail du programmeur. Finalement, la richesse de MXML réside également dans son adaptabilité. Le programmeur peut créer des composants MXML qu'il définit comme une présentation dotée d'un assemblage de composants. Il peut ensuite utiliser ces composants. Ici aussi la propriété de data binding va servir. Elle va permettre de paramètrer le nouveau composant. Ainsi pour notre structure associant la photographie d'une personne avec son nom, nous pourrions imaginer un composant présentant le personnel d'une entreprise, où on passerait en paramètre la source de l'image et le texte à indiquer en dessous (code de la table 4.1.6.2).

Les deux paramètres textuels sont définis comme [Bindable] (lignes 7 à 10) et à la création du composant le nom des deux variables jouera le rôle de méta-donnée que l'utilisateur pourra paramètrer. Ainsi, dans une autre partie de l'application, on pourra utiliser le nouveau composant comme dans l'exemple du tableau 4.1.6.2. Il devra alors être déclaré dans l'en-tête du fichier xml afin de pouvoir être utilisé [11].

[10] Cet exemple est implémenté dans la machine virtuelle.

[11] Ici j'émets l'hypothèse que mon composant est enregistré sous le nom utilisateur.mxml dans le dossier mesComposants à la racine du projet.

```
1   <f:script>
2       <![CDATA[
3
4           [Bindable]
5           public var nom:String = "";
6
7           public function button_click():void{
8               var img:Image = new Image;
9               if(saisieNom.text=="Sandy"){
10                  img.source="sandy.jpg";
11                  boiteParente.addChildAt(img,0);
12                  nom = "Bienvenue Sandy";
13              } elseif(saisieNom.text=="Mathieu"{
14                  img.source="mathieu.jpg";
15                  boiteParente.addChildAt(img,0);
16                  nom = "Bienvenue Mathieu";
17              } else {
18                  nom = "Je ne vous connais pas";
19              }
20          }
21      ]]>
22  </f:script>
23
24  <mx:VBox id="boiteParente" height="90%" width="90%">
25      <s:Label id="zeLabel" text="{'Bienvenue' + nom}"/>
26      <s:TextInput id="saisieNom" text="Utilisateur :"/>
27      <s:Button label="Test" id="zeBouton" click="button_click()"/>
28  </mx:VBox>
```

TAB. 4.10 –

```
<?xml version="1.0" encoding="utf-8"?>
<s:VGroup xmlns:fx="..." xmlns:s="..." xmlns:mx="..." height="90%" widh="90%">

    <f:script>
     <![CDATA[

        [Bindable]
        public var nom:String = "utilisateur par";
        [Bindable]
        public var source:String = "default.jpg";

        public function button_click():void{
            var img:Image = new Image;
            if(saisieNom.text=="Sandy"){
                img.source="sandy.jpg";
                boiteParente.addChildAt(img,0);
                nom = "Bienvenue Sandy";
            } elseif(saisieNom.text=="Mathieu"{
                img.source="mathieu.jpg";
                boiteParente.addChildAt(img,0);
                nom = "Bienvenue Mathieu";
            } else {
                nom = "Je ne vous connais pas";
            }
        }
    ]]>
</f:script>

<mx:VBox id="boiteParente" height="90%" width="90%">
    <s:Label id="zeLabel" text="{'Bienvenue' + nom}"/>
    <s:TextInput id="saisieNom" text="Utilisateur :"/>
    <s:Button label="Test" id="zeBouton" click="button_click()"/>
</mx:VBox>
```

TAB. 4.11 –

```
<?xml version="1.0" encoding="utf-8"?>
<s:VGroup
    xmlns:fx="..."
    xmlns:s="..."
    xmlns:mx="..."
    xmlns:mc="mesComposants.utilisateur.mxml">

    <mc:utilisateur source="sandy.jpg" nom="Sandy" />

</s:VGroup>
```

TAB. 4.12 –

Il existe un type de composant particulier qui a son importance. Il est particulièrement utile dans la représentation personnalisée d'informations et est utilisé extensivement dans le module de recherche de l'interface. Il s'agit des *item renderers* . Ces composants permettent de définir comme ci-dessus la manière dont on va représenter une information mais dans le cadre d'une liste ou d'une collection d'informations. En effet, si l'on a un seul utilisateur à représenter, il est simple d'indiquer que le composant doit utiliser ponctuellement notre nouveau composant. En revanche, pour une liste avec un nombre inconnu d'utilisateurs à présenter, la structure statique du MXML ne semble pas adaptée. On définit alors notre nouveau composant en tant que *<s :itemRenderer>* plutôt que le plus général *<s :Group>*. Ce type de composant peut être transmis à un composant *<s :DataGroup>* qui prend en paramètre une source de données. La source peut être une liste ou une ArrayCollection. Dans les deux cas tous les éléments de la collection de données seront représentés par la structure du itemRenderer. La structure Datagroup peut également prendre une fonction de rendu plutôt qu'un item renderer. Dans ce cas, la fonction qui renvoie un itemRenderer permet de déterminer le Renderer à employer pour l'ensemble de données. Par exemple, nous pourrions déterminer que les résultats d'une recherche peuvent être de différentes natures et créer une fonction qui en fonction du type de données à présenter, renvoie le renderer pour ce type de données en particulier.

Imaginons que notre composant aie été défini comme étendant le composant itemRenderer plutôt que Group (voir table 4.13). Il ne recevrait pas de paramètres spécifiques comme dans le cas du composant mais une donnée de type Object qui constituerait le n^{ime} enregistrement dans l'ArrayCollection ou la liste. Cette donnée sera transmise au travers du paramètre *data* que l'on assigne à nos variables. Nous supposons ici que notre source d'information comprendra un champ «utilisateur» et un champ «source» (lignes 8 et 10).

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <s:itemRenderer xmlns:fx="..." xmlns:s="..." xmlns:mx="..." height="90%" widh=
        "90%">
3
4       <f:script>
5        <![CDATA[
6
7           [Bindable]
8           public var nom:String = data.utilisateur;
9           [Bindable]
10          public var source:String = data.source;
11
12          public function button_click():void{
13              ...
14          }
15       ]]>
16  </f:script>
17
18  <mx:VBox id="boiteParente" height="90%" width="90%">
19      <s:Label id="zeLabel" text="{'Bienvenue' + nom}"/>
20      <s:TextInput id="saisieNom" text="Utilisateur :"/>
21      <s:Button label="Test" id="zeBouton" click="button_click()"/>
22  </mx:VBox>
```

TAB. 4.13 –

Dans ce cas nous pourrions l'utiliser comme dans la table 4.1.6.2. En supposant qu'à un moment donné la variable *donnees* soit populée alors seront affichées une succession d'utilisateurs avec la représentation fixée par l'item renderer.

La définition de l'agencement des composants (*<s :layout>...</s :layout>*, lignes 15 à 17) n'est pas obligatoire mais l'agencemente par défaut proposé par le DataGroup superpose ses composants ce qui n'est généralement pas l'intention du programmeur.

Bien que les exemples ci-dessus restent très basiques, ils suffisent à comprendre l'ensemble des structures utilisées dans l'application et leurs interactions, ce qui démontre bien la puissance et l'adaptabilité

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <s:VGroup
3       xmlns:fx="..."
4       xmlns:s="..."
5       xmlns:mx="...">
6
7       ...
8
9       [Bindable]
10      public var utilisateurs:ArrayCollection = new ArrayCollection;
11
12      ...
13
14      <s:DataGroup dataProvider="{utilisateurs}"
15      itemRenderer="mesComposants.utilisateur">
16          <s:layout>
17              <s:VerticalLayout/>
18          </s:layout>
19      </s:DataGroup>
20
21  </s:VGroup>
```

TAB. 4.14 –

du langage MXML.[12]

4.1.6.3 Présentation de l'outil *FlashBuilder 4*

Jongler entre les langages MXML, ActionScript et PHP pour composer une seule application peut être intimidant et difficile au début. Heureusement, des outils très adaptés sont à disposition, et en particulier FlashBuilder 4 est excellent pour accompagner le développement en Flex. L'IDE est payant (et coûteux), mais la licence peut être obtenue gracieusement pour le personnel universitaire ou les étudiants qui souhaitent utiliser l'outil à des fins académiques.

L'outil est disponible sous deux versions : en tant que plugin pour l'IDE Eclipse et en tant qu'application indépendante. FlashBuilder est construit en Java sur le noyau open source d'Eclipse et reprend l'ensemble de ses fonctionnalités et une bonne partie de ses raccourcis. Nous notons par exemple un mode debug de l'application qui permet à tout moment de controler son exécution, d'identifier les zones de codes problématiques et la valeur des différentes variables à un instant donné. FlashBuilder peut être paramètré pour utiliser le framework Zend afin d'accéder très simplement à l'ensemble des services PHP développés (voir figure 4.6). Pour chacun de ces services il permet de définir le type des différents paramètres et de vérifier la validité des appels de ces procédures. Les services peuvent être testés, leur valeur de retour est évaluée par l'IDE qui propose dans le cas où il reçoit un type qu'il n'identifie pas, de créer le type d'objet correspondant.

L'accompagnement du développement avec la documentation MXML et ActionScript ainsi que les suggestions de complétion (voir figure 4.7) permettent de faciliter la découverte du langage. La composition de l'architecture du programme est également simplifiée. L'IDE permet de créer différents types de fichiers au travers de formulaires simples (voir figure 4.8) : *Component*, *itemRenderer*, class ActionScript ... A la création de ces fichiers, il crée l'entête et les inclusions de librairies nécessaires. Ces inclusions de librairies sont complétées automatiquement au fur et à mesure du développement avec les types d'objet utilisés par le programmeur. Toutes ces fonctionnalités accélèrent considérablement le code dans le courant de son développement mais permettent également à l'utilisateur novice de se familiariser avec la décomposition d'une application Flex et des moyens que le framework met à sa disposition pour arriver à ses fins.

[12] Le site d'Adobe propose (dernière consultation le lundi 30 août) d'excellentes méthodes d'apprentissage pour découvrir leur framework et en particulier un ensemble de vidéos : *Flex in a week* (`http://www.adobe.com/devnet/flex/videotraining/`)

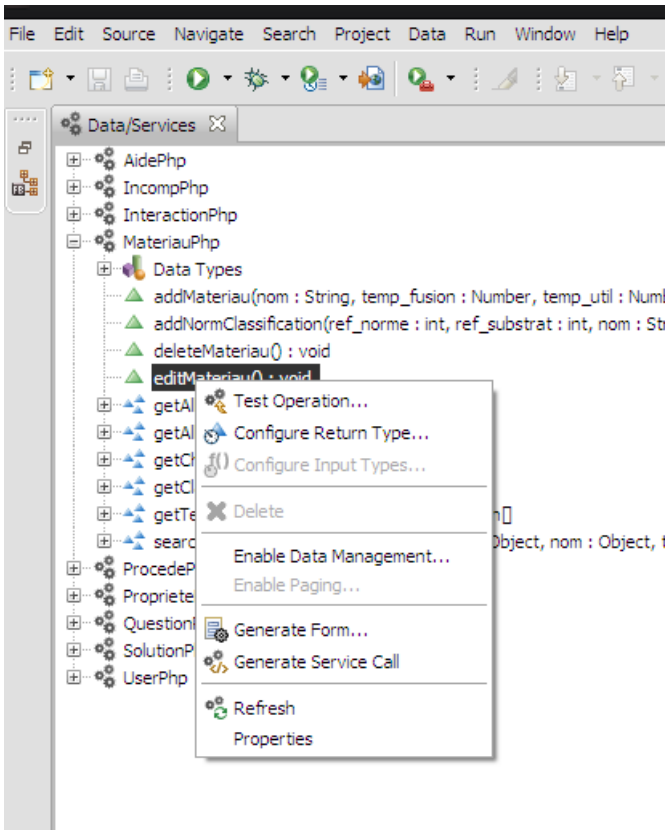


FIG. 4.6 – Opérations sur les services

Finalement, FlashBuilder propose un système WYSIWYG (voir figure 4.9) des interfaces. Le programmeur débutant peut ainsi déposer, redimensionner des composants et éditer leurs différents attributs en marge de la page. Le résultat de ces manipulations n'est pas toujours le plus lisible. Par exemple, les composants sont dimensionnés de manière figée puisque l'éditeur ne sais pas deviner qu'on veut leur donner une taille proportionnelle. Afin d'épouser le mode de pensée le plus communément employé par les développeurs d'application web, MXML s'est doté d'un attribut associé à tous les composants visuels : *state*. On peut au travers d'un composant *<s :state>* définir des états. Leur utilisation dans les attributs de l'application permettra de dire qu'un composant appartient ou est visible dans un état donné. Le fonctionnement des états permet en définitive de présenter des pages. L'éditeur WYSIWYG proposera ainsi les interfaces en fonction de l'état dans lequel elles se trouvent. Cette vision est intéressante lorsque l'on travaille avec un nombre restreint de pages. Elle rend néanmoins le code MXML moins lisible. En effet, il devient difficile de visualiser autrement qu'au travers de l'éditeur graphique, la représentation qu'aura l'application avec les définitions d'états qui apparaissent pour chaque composant visuel. L'éditeur graphique est également limité puisqu'il ne présente pas un composant déclaré initialement comme invisible et que l'on fait apparaître de manière procédurale. Il ne permet pas de se rendre compte de la dynamique associée à l'interface au travers de l'ActionScript comme on l'aurait dans un fichier MXML qui présente le script associé. En revanche, dans un premier temps, pour découvrir les différents composants et appréhender le couple MXML/ActionScript, l'éditeur graphique est intuitif et complet. Il suffit également pour de plus petits développements avec un nombre restreint de composants.

4.2 Choix d'implémentation

Toutes les difficultés des nouvelles exigences de l'interface ne peuvent être résolues au travers du seul choix de technologies. Certains choix d'implémentation ont permis de répondre à des difficultés spécifiques qu'il convient de détailler.

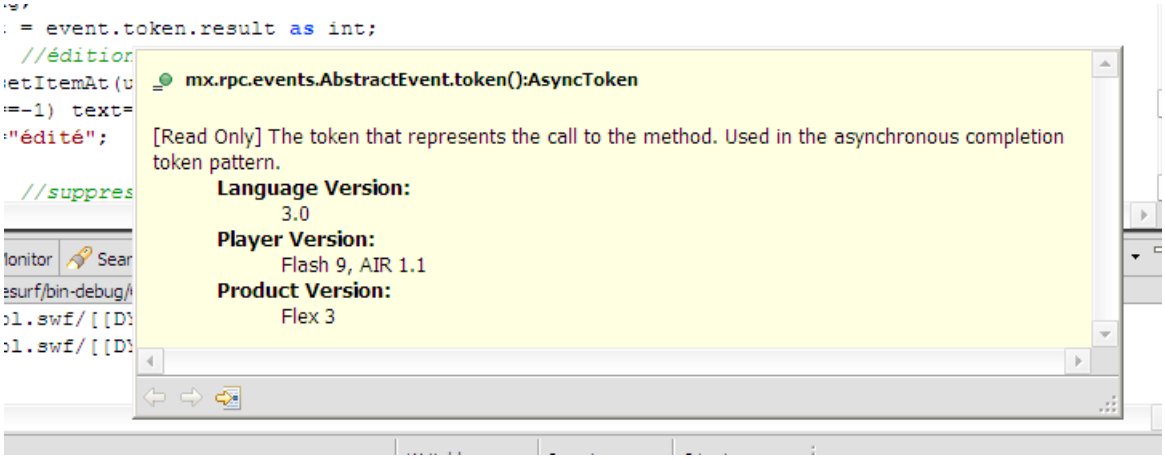


FIG. 4.7 – Documentation des langages

4.2.1 Abandon de la double-validation

Une première difficulté qu'il nous a fallu adresser était le fait que la double-validation existante n'était pas contrôlée par l'interface au travers de quelques colonnes dans les tables de la base de données mais était véritablement gérée par la base de données en elle-même au travers d'une interaction complexe entre différents triggers et fonctions. Ce processus de double-validation n'est pas à supprimer au complet pour deux raisons importantes. La première, c'est que l'interface de Zope en dépend et que bien qu'il est question de l'abandonner il serait dommage qu'on ne puisse plus s'en servir sitôt la seconde interface installée ne fusse que pour continuer à présenter différemment les relations familiales de la base de données. Ensuite, cette structure est complexe mais efficace et adaptable à n'importe quelle interface. Il n'est pas dit qu'un jour cette fonctionnalité ne sera pas de nouveau requise. Il pourrait, par exemple, être envisageable d'introduire un niveau d'utilisation intermédiaire de l'application permettant de suggérer des modifications sans qu'elles ne soient directement actées dans la base de données et que seul le niveau super-administrateur puisse valider les modifications proposées.

Nous pouvions aborder la double-validation de deux manières différentes. La première piste, partiellement explorée, consistait à utiliser les triggers artificiellement en contre-validant par programmation les demandes de modifications lors de leur introduction. Pour les cas simples tel que des insertions de procédés ou d'interactions, cela ne posait pas de problème. En revanche, certaines insertions réclamées par le client sont interdépendantes. Ainsi, la qualification d'un matériau ne peut être introduite sans en avoir l'identifiant. Le fonctionnement de ces formulaires plus complexes est séquentiel. On ajoute les enregistrements dépendants après la bonne insertion des autres (en fonction des contraintes référentielles). Or, le système de triggers mis en place ne permet la validation d'une qualification que lors de la contre-validation de l'insertion de son matériau. Comme chaque insertion est commanditée par l'interface, celle-ci aurait dû confirmer, après une succession de demandes d'insertion, celle du parent inséré préalablement. Cette démarche aurait rendu l'architecture de la nouvelle interface dépendante de l'organisation des triggers

La seconde option était de désactiver les triggers en contrôlant les modifications avant chaque insertion et de les rétablir après. Une fonction existait déjà pour désactiver les triggers sur un ensemble de tables. En effet, lors de la validation ou de la suppression d'une information les triggers étaient déjà désactivés le temps de la modification. Le choix de contourner les triggers s'est donc imposé. C'est la solution qui réclame le moins d'adaptations de l'implémentation et donc si toutefois la structure de la base de données venait à évoluer pour qu'on abandonne définitivement les triggers, seules deux lignes[13] de l'implémentation dans une fonction en PHP sur le serveur seraient à modifier. En attendant, ces deux lignes impliquent deux requêtes supplémentaires requises à chaque insertion, édition et suppression.

[13]Il faut supprimer l'appel à *disable_state_changes* avant les modifications et à *enable_state_changes* après.

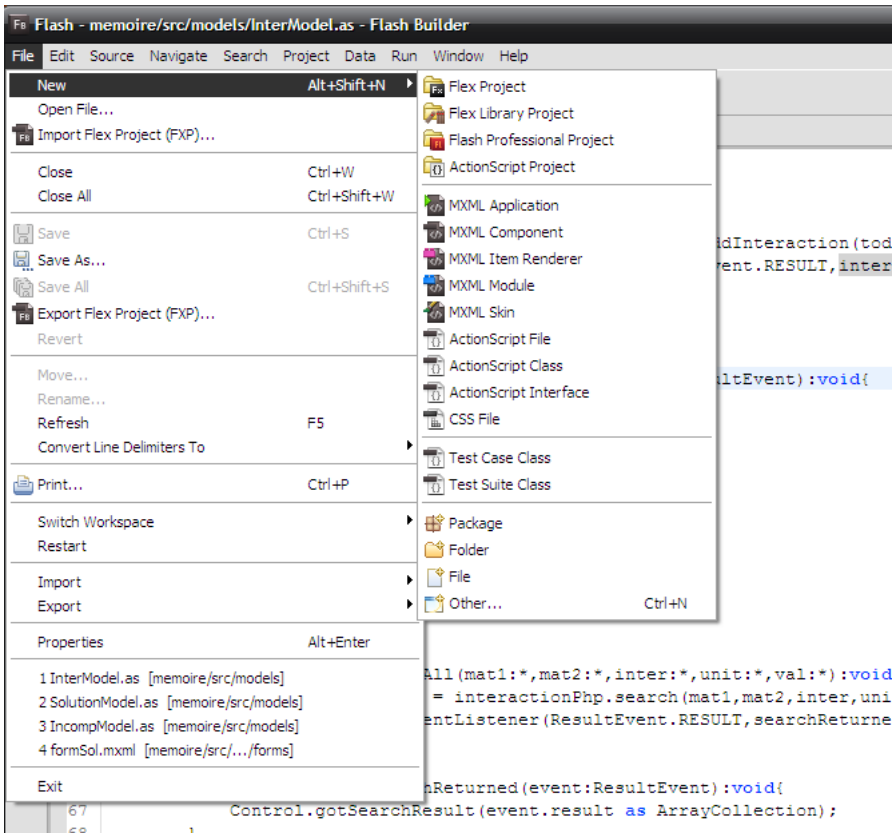


FIG. 4.8 – Ensemble des types d'objets que l'on peut créer automatiquement

4.2.2 Aplatir les hiérarchies parentales

De la même manière que l'interface présente la base de données en faisant abstraction des différents états de ses enregistrements, elle devra également en présenter les données comme si elles ne faisaient pas l'objet d'une hiérarchie familiale. Cette relation parentale intervient dans trois tables : `PROCEDE`, `MATERIAU` et `PROPRIETE`. Les relations ne seront pas traitées de la même manière d'une table à l'autre parce que sémantiquement elles ont des implications différentes.

4.2.2.1 La difficulté de l'héritage

On retrouve différents types de relations parentales d'une table à l'autre. Dans tous les cas, elles définissent qu'un enfant *est du type de* son parent mais la relation parentale ne se répercute pas de la même manière dans tous les rôles que joue le type d'entité. Ainsi, une incompatibilité ou une solution faisant intervenir un matériau se transmet à tous ses enfants. En revanche, une qualification portant sur une propriété ne signifie pas que l'objet de la qualification porte également sur tous ses enfants. Si l'acier présente une incompatibilité avec un matériau x, il est incompatible avec tous les fils de x. Par contre, la propriété couleur qui qualifierait un matériau ne signifie pas que le rouge, le bleu et le transparent qualifient également matériau. Par suite, les enfants d'un type d'entité, n'entrent pas dans toutes les relations de la même manière. De ce fait, un comportement différent sera adopté en fonction du domaine d'application pour déterminer si oui ou non il faut remonter la hiérarchie familiale pour s'approprier une caractéristique ou si elle n'a pas d'implication dans cette relation. Ces rapports se complexifient d'autant que les trois tables concernées par ce type de relation sont au coeur du schéma de la base de données et que donc elles interviennent dans de nombreuses relations avec d'autres tables et par leur intermédiaire, entre-elles.

Nous pouvons traiter la relation familiale de deux manières différentes. Soit nous l'abandonnons et considérons les enfants et les parents comme des enregistrements simples, soit nous nous efforçons à présenter les résultats en ayant aplani leurs relations. La première solution est plus simple mais problé-

matique dans bien des cas. Elle implique une perte d'information. On perd la notion qu'un enregistrement est parent d'un autre. Au regard du moteur d'inférence qui utilise l'héritage pour définir les qualités des différentes entités, cette abstraction est problématique. A l'insertion l'utilisateur associera des éléments qu'il pense être des feuilles sans se rendre compte de leur portée et qu'ils vont donc attribuer des qualités cachées par une hiérarchie qu'il ignore.

Si l'on adopte la seconde solution, certains résultats devront être aplanis. En effet, si la base de données ne présentait plus de familles, elle ne présenterait plus que des «enfants» auxquels on aurait attribué toutes les propriétés héritées. Les résultats d'une recherche présenteraient donc des relations qui ne correspondent pas à un enregistrement en base de données mais qui sont le fruit d'un héritage. Nous ne présenterions, par exemple, non pas l'incompatibilité entre la *Famille Ni-P (270)* et la *Famille Fe-C-Ni (104)* mais un ensemble d'incompatibilités : le produit cartésien de tous leurs enfants au pied de l'arborescence familiale (tous les enfants qui ne sont pas parents eux-même). SI nous voulions adopter cette solution, sans modifier la base de données, on devrait créer ces enregistrements «artificiels». A ce titre, ils ne pourraient être édités ou supprimés. Cela représenterait une limitation pour l'interface présentant ces données. De même, certains enregistrement réels n'apparaîtraient plus jamais (les enregistrements des familles). Il ne serait donc plus possible de les modifier non plus.

Dans le cadre de notre développement, on a utilisé tantôt la première, tantôt la seconde solution. Le choix de présenter des informations sous une autre forme que telles qu'elles sont enregistrées limitera certaines opérations. Les familles n'existant pas au regard de la nouvelle interface, elles ne pourront être éditées sans atteindre à l'intégrité de la base de données. A l'inverse, l'utilisation de la première solution entraîne une perte d'information puisqu'on ignore la relation familiale que l'on possède pourtant.

Certains enregistrements dans la base de données posent problème. En effet, l'objectif de l'aplatissement de la représentation des données vise d'une part à rester cohérent avec une insertion sans hiérarchie, mais d'autre part à permettre la mise en évidence des enregistrements inutiles ou corrompus. C'est le cas, par exemple, des familles sans héritiers. Il apparaît, dans certains résultats de recherche, des interactions entre un matériau et aucun autre tout simplement parce que le deuxième matériau référencé en base de données n'a pas au bout de son arborescence d'enfants qui ne soient pas parents. Le parent ne présentant comme intérêt que la transmission de ses propriétés à une couche ou un substrat véritable, ces familles entières sont inutiles dans la base de données. C'est le type d'incohérence qu'on espérait mettre en évidence au travers de la nouvelle interface. Les résultats de recherche seront présentés comme s'il n'y avait plus de famille. A ce même titre, aucune modification des familles n'est possible au travers de l'interface. Les nouveaux enregistrements seront conformes à une représentation plane[14].

4.2.2.2 Les propriétés

Les propriétés possèdent une hiérarchie familiale qui permet de représenter des relations parent-enfant telles que le fait que la propriété *rouge* soit une sous-propriété de la propriété *couleur*. Ainsi, un matériau qualifié par la propriété rouge est qualifié également par la propriété parente : couleur. A l'affichage, un aplatissement aurait été superflu. Lorsqu'on présente un matériau de propriété *rouge*, il n'est pas intéressant de préciser qu'il est également de propriété *couleur* bien que ce soit sémantiquement correct. La propriété la plus intéressante est en définitive la plus spécifique. Les familles de propriétés n'ont pas d'autres implications et n'interviennent pas dans des relations complexes. En effet, un test ne met en évidence que la propriété à laquelle il est directement associée. Le fait qu'un test permet d'identifier le parent d'une propriété n'implique pas qu'il identifie ses enfants. Pour ces raisons, nous nous sommes contentés d'ignorer les relations parentales dont les propriétés faisaient l'objet ; elles sont traitées comme des enregistrements simples. A l'insertion comme à la présentation, on peut associer, dissocier et manipuler indifféremment un enregistrement enfant et un enregistrement parent.

4.2.2.3 Les matériaux et procédés

Les matériaux, contrairement aux propriétés, présentent de nombreuses relations avec les autres tables ainsi qu'un rapport familial qu'on ne peut se contenter d'ignorer. Il a donc fallu décider de comment nous allions aplanir leurs relations familiales. Ces hiérarchies s'étendent parfois sur une dizaine de niveaux. A l'insertion, toute association à un matériau (interaction, incompatibilité...) ne sera dorénavant possible que pour des matériaux enfants. Ne seront présentés dans les résultats de recherche que des matériaux

[14]Ils sont enregistrés en tant qu'enfant du parent racine : 1.

enfants répondant aux critères de recherche également. De ce fait, une incompatibilité ne fera peut-être plus intervenir un seul matériau et une seule couche mais l'ensemble de leurs enfants entre-eux. Une interaction entre deux enregistrements de matériaux devra être représentée comme une interaction entre deux ensembles de matériaux. Une qualification qualifiera un ensemble de matériaux tandis qu'une application s'appliquera à un ensemble de matériaux,...

On appliquera aux familles de procédés la même démarche que pour les matériaux. Ne seront présentés en résultat de recherche et autorisés en insertion que des procédés enfants.

4.2.2.4 Common.php

L'ensemble des fonctions utilisées afin d'aplatir une hiérarchie familiale ont été regroupées dans un même fichier importé dans les différents services PHP. Dans l'éventualité où les relations familiales de la base de données venaient à être abandonnées seules ces fonctions seraient à modifier (supprimer leurs différents appels ou mettre en commentaire leur contenu).

Pour chaque démultiplication d'un ensemble d'enregistrements on appelle une procédure qui aplatit, en fonction d'un champ de référence, un type de donnée spécifique : *flattenOnProc*, *flattenOnMat*, *flattenOnProp*. Elles comportent toutes la même structure récursive qui se présente comme suit :

flattenOnProc($connec,$result)

Arguments :

$result est un tableau contenant un champ *ref_proc* et un champ *val_state* référençant un procédé dans la table `PROCEDE`.

$connec est un objet de type Connec (*Connec.php*).

Valeur de retour : La fonction renvoie le tableau initial non modifié pour les enregistrements du tableau ne référençant (*ref_proc*) pas un procédé «parent». Sinon, les éléments «parent» ont été dupliqués autant de fois que le procédé qu'ils référencent n'a d'héritiers[15] dans la table `PROCEDE`. Ces nouveaux enregistrements ont *val_state* égal à -1[16].

```
1   function flattenOnProc($connec,$result){
2       if(count($result)!=0){
3           $current = $result[0];
4           $isParent = $connec->query(
5               "select parent from procede where id_proc=".$current->ref_proc);
6           if(count($result)==1)
7               $part2 = array();
8           else
9               $part2 = flattenOnProc(array_slice($result,1));
10          if($isParent[0]->parent){
11              $ref = $current->ref_proc;
12              $children = $connec->query(
13              "select id_proc from procede where ref_parent=".$ref);
14              if(count($children)==0)
15                  $part1 = array($current);
16              else{
17                  foreach($children as $child){
18                      $line = clone $current;
19                      $line->ref_proc = $child->id_proc;
20                      $line->val_state = -1;
21                      $part1[]=$line;
22                  }
23                  $part1 = flattenOnProc($connec,$part1);}}
24          else
25              $part1 = array($current);
```

[15] J'appelle héritier d'un procédé les feuilles de l'arborescence familiale à laquelle il appartient.

[16] C'est cette valeur qui permet d'identifier que l'enregistrement est un enregistrement hérité et qu'il ne correspond pas à un enregistrement en base de données. Ce champ permet à l'interface de savoir si l'enregistrement est éditable et supprimable ou non. Pour rappel (rétro-ingénieurie), *val_state* indique normalement l'état de l'enregistrement dans la base de données.

```
26              return array_merge($part1,$part2);}
27          else return $result;
28      }
```

4.2.3 Un nouveau mode de recherche

Il existait déjà un mode de recherche «Navigationnel» dans la première interface. Elle permettait, partant de la liste des racines des familles de déployer les éléments de fils en fils en présentant à chaque fois un ensemble de caractéristiques pouvant être consultées puis éditées ou supprimées.

Il nous a donc fallu déterminer un nouveau mode de représentation des informations qui permettraient de mettre en valeur les relations que l'on retrouve dans le schéma conceptuel (détaillé dans le chapitre sur la rétroingénieurie). L'objectif sera alors de pouvoir partir d'un procédé et de pouvoir visualiser l'ensemble des enchaînements dont il peut faire partie. Ensuite, depuis l'enchaînement, il est important de pouvoir découvrir les informations des procédés qui le concernent et ainsi de suite. Cette navigation peut en fait se résumer par le graphe :reproduit figure 4.10. On navigue au moyen des flèches de noeud en noeud au fil de leurs relations. Les informations qui caractérisent directement le noeud sont l'ensemble des informations qui concernent ce type d'entité ainsi que les informations des noeuds qu'il comprend et qui lui sont associées. Ainsi, les normes auxquelles sont liées (au travers de la relation «classification») un matériau sont des informations directement accessibles dans le noeud matériau. On ne sait pas accéder à une norme autrement qu'au travers d'un noeud matériau.

Dans le mode de navigation de l'ancienne interface on partait nécessairement d'une liste exaustive de toutes les racines des arborescences familiales, après quoi on pouvait déployer leurs différents descendants successifs. Ici, il nous faut déterminer de quel noeud va partir la recherche et comment on va définir l'ensemble des points d'ancrages possibles. La navigation va se faire en partant d'un résultat de recherche. On n'aura pas une liste de matériaux racines par exemple mais un ensemble de feuilles répondant aux critères d'une recherche donnée et à partir desquelles on ne déploiera pas des matériaux «enfants» mais des caractéristiques étendues. Au sein du graphe, certaines relations ont la priorité sur d'autres. Tout comme dans le schéma conceptuel de la base de données on a une dépendance de certaines tables par rapport à d'autres. Au niveau du domaine d'application, certains types d'entités ressortent comme étant plus significatives que d'autres. On note notamment les types d'entités qui font l'objet d'une relation familiale ou ceux que l'on peut choisir d'insérer dans la partie «insertion» de l'application. C'est pourquoi, c'est de ces noeuds centraux que partiront la recherche. On peut effectuer une recherche directe sur les types d'entités représentés par un noeud jaune : `MATERIAU`, `PROCEDE`, `PROPRIETE`, `INTERACTION` et `INCOMPATIBILITÉ`.

Les deux nouveaux modes de recherche sont donc étroitement liés : la recherche navigationnelle présente les résultats de la recherche directe et, de son côté, la recherche directe permet d'accéder à des noeuds depuis lesquels on peut effectuer la recherche navigationnelle. On a donc deux manières d'accéder à une propriété par exemple. On peut soit la rechercher directement au travers du formulaire de recherche ou soit, si on ne la connait pas directement et qu'on sait qu'elle qualifie un certain matériau, on peut naviguer vers elle depuis un autre noeud de la recherche navigationnelle.

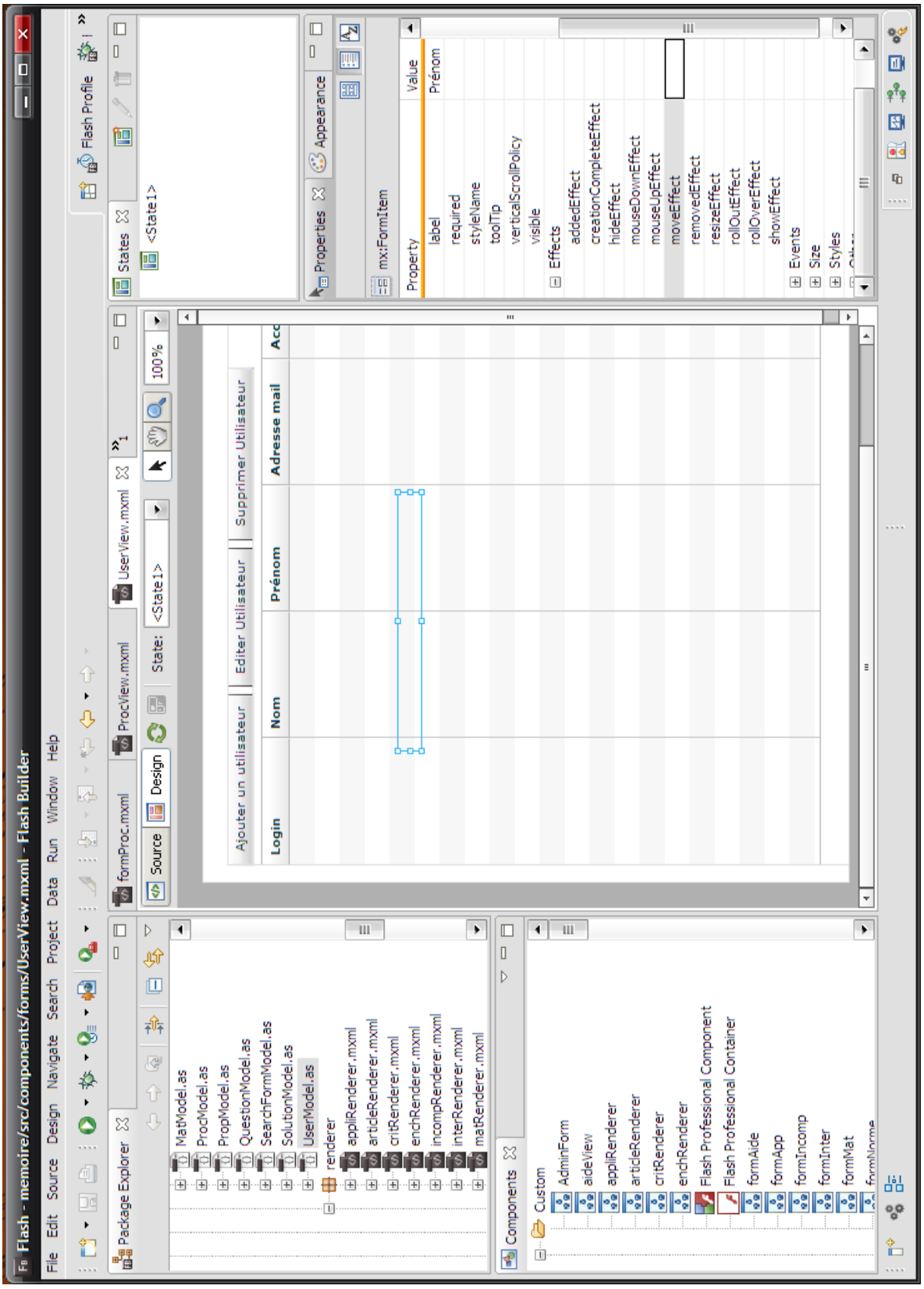


FIG. 4.9 – Éditeur d'interfaces MXML *WYSIWYG*

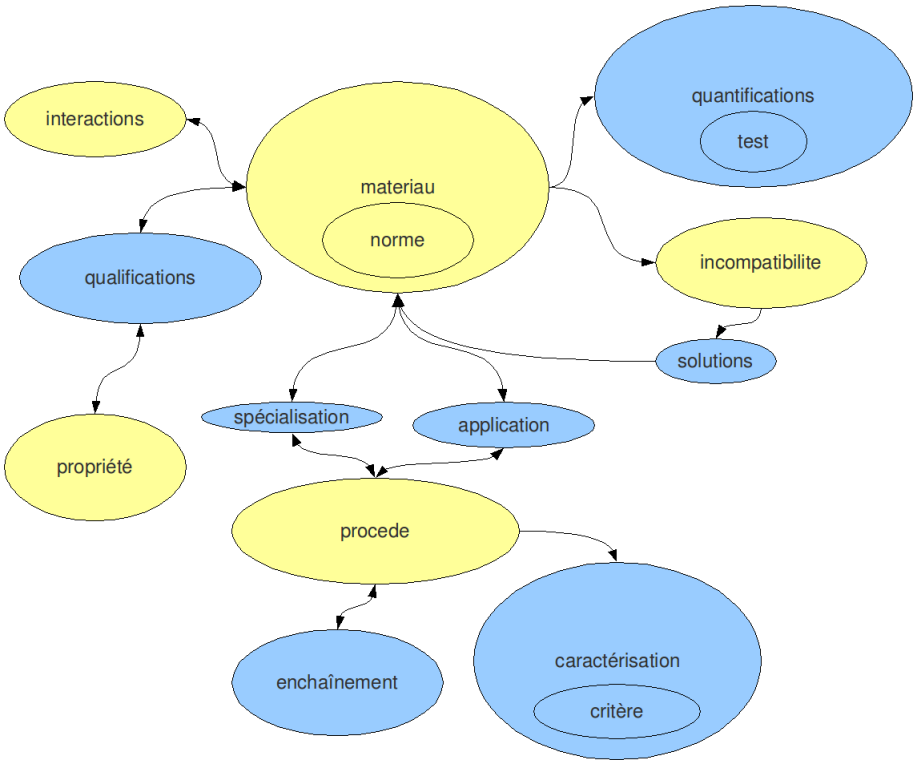


FIG. 4.10 – Graphe de navigation dans les résultats

4.3 Architecture

Il semblait important de décider des points plus délicats du développement avant d'aborder l'architecture d'une interface. En effet, l'architecture doit être au service de ces choix décisifs afin de faciliter leur développement. En adoptant la démarche inverse (de fixer l'architecture du logiciel avant de déterminer les choix d'implémentation plus délicats), on prend le risque de devoir adapter des choix, pourtant déterminants, à l'architecture plutôt que l'inverse. Une fois les questions les plus délicates résolues, nous nous sommes donc attellés à la construction d'une architecture.

L'élaboration de l'architecture d'un programme consiste en la démarche inverse de la rétro-ingénieurie expliquée au chapitre 1. En partant d'une conceptualisation de haut niveau, on va affiner progressivement notre modélisation pour obtenir une modélisation de l'implémentation où les éléments du schéma final, ont un équivalent immédiat dans le code du programme. Evidemment, différents modèles d'analyse existent. Ils mettent chacun l'accent sur un point de vue du code. Comme tout modèle, par définition, ils font abstraction de certains éléments et, en fonction de ce qu'ils choisissent de présenter par rapport à d'autres, ils donnent un point de vue différent face à la structure du programme. Il est donc important de choisir un type de modélisation qui rend bien compte des spécificités du code.

Dans la première partie de ce mémoire, nous avions réalisé ce type d'exercice avec l'interface Zope. Nous avions envisagé une modélisation au travers d'UML avant de rendre compte de certaines spécificités de Zope qu'il ne serait pas possible de représenter sans ambiguïté. Nous sommes de nouveau face à un problème de modélisation, et cette fois encore, le langage en jeu, est orienté objet. Il y en a même trois : ActionScript est orienté objet, PHP également et finalement MXML peut être traduit en ActionScript. Toutefois, contrairement au framework Zope qui introduisait des concepts singuliers dans les relations entre les objets, ActionScript, PHP et MXML ne font appel qu'à des relations que l'on peut représenter au travers de l'architecture UML.

UML ne propose pas un type de modélisation mais de nombreuses modélisations possibles afin de présenter le comportement, la structure et les interactions du programme. Certaines de ces modélisations ont un intérêt secondaire dans le cadre de notre application. Les seuls échanges distribués étant entre le composant client et serveur et, ces interactions étant fort simples, rares sont les échanges qui mériteraient une modélisation. De même, un diagramme de composants pourrait sembler superflu dans la mesure où seul le composant serveur offre une interface de services que seul le client emploie. De plus, nous avons déjà décrit les protocols employés et l'organisation de cette structure autrement dans la présentation de l'organisation de Flex. En revanche, il serait intéressant de présenter la manière dont s'agencent les classes au sein du programme.

En effet, du fait des différents types de composants et de fichiers que l'on peut retrouver dans une application Flex, le code source est découpé en de nombreux fichiers avec des rapports de dépendance et d'utilisation fort différents. Comme nous l'avions mentionné, la partie «modèle» de l'application apparaissait assez clairement au travers de l'ensemble des classes auto-générées et de leur utilisation par des classes en ActionScript pur. La structure de la «vue» de contenant en contenu est illustrée plus intuitivement au travers de la présentation visuelle de l'interface plus loin. En revanche, l'organisation du «contrôleur» que l'on disait diffus au sein de l'application et qui gère les transitions entre les vues et les recours au modèle n'a pas été décrite. Elle joue pourtant un rôle essentiel dans l'interface. Afin de bien la mettre en évidence, les diagrammes de classes UML[17] sont les plus adaptés. Ils présentent les relations entre les différentes classes d'objets en jeu. Dans le cadre de cette modélisation nous considérerons les composants MXML sous la forme des classes ActionScript qu'ils produiraient. En revanche, nous ferons abstraction des nombreuses déclarations de variables stucturelles définissant les éléments de la vue qui chargeraient les schémas sans apporter d'information supplémentaire. Retenons simplement que toutes les classes n'appartenant pas au package «models» sont dotées, en plus de leurs attributs et méthodes, d'une structure MXML définissant la représentation visuelle de l'objet.

4.3.1 Modélisation du composant Client

L'architecture globale de l'interface de l'utilisateur part d'un fichier racine «Control.mxml» qui constitue le point de lancement de l'application. La compilation de ce fichier produit le fichier Flash qui est chargé lors du démarrage de l'interface au complet. C'est pour cette raison que c'est le rapport entre tous

[17] Une explication de l'élaboration et de la syntaxe de ces diagrammes est donnée en annexe (page 121)

les éléments de l'application et ce noyau que nous allons décrire pas à pas [18]. Celles-ci sont dotées d'attributs et d'opérations, elles-même caractérisées par une visibilité[19]. La «visibilité» d'un attribut détermine quels autres objets de l'application peuvent y accéder. Un attribut privé («private») ne peut être ni lu ni modifié par aucune autre instance d'objet du programme. Seules les méthodes propres à l'objet qu'il décrit pourront l'utiliser. Une classe étendant une autre classe n'aura pas accès aux propriétés privées de celle-c. En revanche, elle accèdera tout de même à ses attributs et méthodes protégées («protected»). Enfin, toute déclaration publique («public») peut être utilisée depuis n'importe quelle partie du code du programme.

Les diagrammes de classes seront également l'occasion de présenter certaines fonctions particulières et la signification de certains attributs.

4.3.1.1 Les classes impliquées dans l'insertion d'un matériau

Les figures 4.11 et **??** présentent l'ensemble des classes interagissant pour présenter et effectuer l'insertion de matériaux. La classe MatView compose la classe Control désignant l'application principale. Une instance de MatView présente et coordonne les opérations d'insertion de matériaux. MatView est caractérisée par un attribut de type MatModel qu'elle utilise donc et par un ensemble de tableaux dont les données seront garnies par le modèle. Les attributs suivants indiquent la position du curseur dans une grille d'information donnée. L'ensemble des méthodes présente très bien l'aspect contrôleur de notre application qui réagit aux actions de l'utilisateur sur la vue. La classe MatView est elle-même composée par un ensemble d'autres classes qui permettent de représenter l'abstractioin d'un formulaire de qualification (formQuali), de quantification (formQuanti), de matériau (fromMat), de normes (fromNorme), ou encore d'applications (formAppli). Ces différentes classes sont à peu de choses près toujours de la même forme et proposent un ensemble de méthodes similaires : vider le formulaire (purge), vérifier qu'il est valide (isValid) ou préremplir son contenu (setContent). C'est une structure que l'on retrouvera pour tous les types d'entité insérables. L'ensemble des diagrammes présentant l'insertion des autres types d'entité sont reprise en annexe (page B).

Notons tout de même le contraste entre le développement Flex où la plupart des attributs sont publics et de nombreuses méthodes sont privées à l'inverse de ce que l'on rencontre souvent en Java par exemple où des méthodes publiques permettent l'accès aux attributs privés de façon contrôlée. En flex, on ne rencontre que peu de «setters» et de «getters». Les tutoriaux ne font pas état des mêmes bonnes pratiques qu'en Java où il faut restreindre au maximum la visibilité des attributs. La définition d'un tag d'un composant MXML passe nécessairement par un attribut public. Donc tous les composants MXML se retrouvent avec un ensemble d'attributs publics. De même, un ArrayCollection ne peut être la source de databinding que s'il est défini comme public. De ce fait, bon nombre d'attributs dans toutes les classes seront publics. En revanche, les fonctions représentant souvent une réaction spécifique à l'interface du même composant, elles ne doivent pas pouvoir être sollicitées depuis n'importe où dans le programme et sont déclarées privées.

Il faut noter, qu'au travers de cette structure, Flex impose un environnement de programmation un peu moins sûr. En effet, on peut modifier des données du modèle depuis n'importe où dans l'application sans aucun contrôle de la manière dont ces modifications sont faites.

4.3.1.2 Les classes impliquées dans l'aide et la recherche

La figure 4.13 présente l'ensemble des classes impliquées dans la gestion de l'aide et celles impliquées dans la recherche.

La classe Control a finalement peu de méthodes propres. Elle est davantage caractérisée par un nombre important d'attributs puisque ainsi que décrit par l'architecture du modèle MVC utilisé ici, le contrôleur (dont Control fait partie) instancie les vues et les modèles. Il joue le rôle de coordinateur entre l'utilisateur et la présentation et donc nécessite un accès à celles-ci. Control est le seul objet à utiliser la classe sideList. Une instance de cette classe correspond à une liste couplée à un champ de texte dont la présentation est faite dans un pannel. Les instances de cette classe constituent les panneaux latéraux de recherche rapide pour compléter les formulaires. La classe est caractérisée par une liste d'éléments à présenter, un champ de texte à précompléter et un entier type indiquant le type de données concernées.

[18] Tous les diagrammes de classes présentés dans cette section ont été créés à l'aide de l'outil Visual Paradigm.
[19] Un attribut ou une opération peut être déclarée «publique», «privée» ou «protégée».

Parmi les quelques méthodes de controle apparaissent les méthodes gérant les apparitions/disparitions de ces instances : showMatList, hideSideLists,... Ces méthodes permettent de modifier les attributs privés désignant la visibilité de ces panneaux : matList, procList, etc.

Une instance de la classe aideView représente un outil de présentation et de manipulation de l'aide. Elle utilise à cet effet la classe aideModel dont les attributs reprennent l'ensemble des données de l'aide et dont les méthodes permettent de les charger en mémoire et de les modifier. Les éléments de l'aide son représentés au travers d'instances des classes rubriqueRenderer et articleRenderer. Ces classes étendent la classe itemRenderer et prennent en paramètre une donnée data. En dehors de ce paramètre ces classes ne font qu'un travail de présentation. Elles sont strictement du domaine de la vue.

Une autre classe du même type est questionRenderer. Cet itemRenderer définit la présentation d'une question. L'attribut questions de la classe Control va être présenté au travers de la classe questionRenderer. C'est sa seule utilisation dans l'application.

La recherche est finalement simple. La classe specSearch permet de créer un espace de recherche et de populer une zone d'un navigateur : resultNavigator avec les résultats de la recherche. La classe fait usage de la classe d'objets SearchFormModel qui comme son nom l'indique est le modèle associé à une recherche. Une instance de cette classe effectuera les communications serveur nécessaires afin de garnir ses attributs publics des données présentées dans les vues qui lui sont liées.

4.3.1.3 Les classes permettant de proposer la gestion des utilisateurs

Le diagramme des classes permettant la gestion des utilisateurs se trouve à la figure 4.14. On retrouve de nouveau la classe de modèle des données mais le formulaire d'insertion n'a pas été dissocié de la classe principale UserView. Il ne sert que dans cette partie de l'information.Il était donc plus intéressant de garder toute la vue dans l'objet principal UserView.

4.14

4.3.2 Modélisation des services proposés par le serveur

Plus importante encore que l'architecture de l'interface, il y a l'architecture du serveur puisque celle-ci peut servir d'autres applications que notre développement en Flex. Les méthodes proposées par les différentes classes de services permettent d'accéder aux données utiles à l'interface mais cette structure, en PHP, est indépendante de toute présentation des données. Afin de continuer à être utilisable par notre composant client tout en le modifiant, le composant serveur doit s'assurer que les différentes classes de services ainsi que les services qu'elles proposent respectent deux conditions :

1. l'intitulé des services et la déclaration de leurs méthodes restent inchangées
2. les modifications apportées ne modifient pas les spécifications des services proposés à l'interface.

En dehors de ces conditions, on peut réorganiser, réimplémenter, et transformer le composant serveur de l'application pour le maintenir à jour, offrir d'autres services ou déservir de nouveaux clients sans affecter le composant client en Flex qui en dépend. L'architecture du composant est donc conçue afin de maximiser sa réutilisabilité.

L'ensemble des services est l'intermédiaire entre une interface et la base de données. Or, ces deux composants reprennent finalement les mêmes grands concepts agencés autour de types d'entités clés du domaine d'application : matériaux, propriétés, procédés, ... Il semble donc tout naturel, pour les classes qui vont permettre le pont entre ces deux composants (interface et base de données), de garder cette même découpe logique (voir diagrammes 4.15 et 4.16). L'ensemble des méthodes permettant de modifier ces grands objets du domaine de l'application sont représentés par les classes `MateriauPhp`, `ProcedePhp`, `ProprietePhp`, `IncompPhp` et `InteractionPhp`. Ces classes de services vont offrir la possibilité d'ajouter, éditer ou supprimer un élément de la table de même nom (la table `propriete` pour la classe `ProprietePhp`) mais également d'effectuer des recherches dans cette table et les manipulations sur des tables annexes. Par exemple, la classe `ProcedePhp` va également proposer toutes les opérations sur les tables `Critere`, `Application` et `Enchainement`.

Trois autres types de services ne concernent pas le domaine d'application et vont rendre possible les opérations sur une table particulière : `UserPhp` avec la table `utilisateur`, `QuestionPhp` avec la table `question` et `AidePhp` pour `aide`.

Toutes ces classes ont besoin d'une connexion à la base de données. De plus, elles vont toutes utiliser le même type d'opération. Afin de ne pas dupliquer le code de cette connexion à la base de données nous

avons rassemblé dans une classe utilisée par toutes les autres les méthodes d'accès à la base de données. Toutes les classes de services utilisent la classe `DBConnec`. Tous les appels à la base de données passent par cette classe qui va se charger de la fermeture des curseurs, de la désactivation des triggers quand c'est nécessaire, ... Grâce à cette découpe, si un jour l'activation et la désactivation des triggers devait être modifiée, nous n'aurions qu'un rapide changement à apporter dans cette classe exclusivement.

La connection à la base de données requière un ensemble de paramètres de connexion : son adresse, un nom d'utilisateur et son mot de passe. Ces informations ont été enregistrées dans un fichier de configuration *db.conf* indépendant qui permet de rapidement paramétrer l'application lors de l'installation. A l'avenir, ce fichier de configuration pourrait s'enrichir et contenir d'autres informations destinées non pas à la base de données mais à une toute autre fonctionnalité. Dans ce cas, il n'est pas intéressant que notre classe de connexion à la base de données se charge de lire le fichier elle-même. Elle utilise plutôt la classe `Config` qui extrait du fichier de configuration les paramètres de la connexion que l'on retrouve en attribut. Si, à l'avenir, on avait plusieurs fichiers de configuration, que certains paramètres venaient à concerner plutôt l'une ou l'autre classe utilisatrice, ou que le format du fichier de configuration était modifié, cette isolation des méthodes liées à sa lecture permettrait de modifier une seule fois la classe `Config` pour répercuter ce changement sur toute l'application.

Il existe deux fichiers de code en PHP qui ne sont pas des classes en tant que telles et qui forment en quelque sorte des «librairies». Ces fichiers proposent un ensemble de méthodes qui peuvent être importées où elles sont utiles.

Le premier groupe de méthodes se trouve dans *Common.php*. Ces méthodes permettent d'aplatir les familles dans un tableau de données. Elles ont été isolées du reste du code parce qu'elles servent dans toutes sortes de classes de services. Si, à l'avenir, une modification à la hiérarchie familiale venait à être apportée, seul ce fichier serait à modifier.

Un autre ensemble de méthode a été regroupé dans le fichier *Session.php*. Lui non plus n'est pas une classe, et pour cause, il opère sur les variables de session. Ces variables sont à PHP ce que les variables globales sont à d'autres langages de programmation puisqu'elles sont rattachées au navigateur qui les rend disponibles quelle que soit la page consultée pour un site donné. Ces variables ne peuvent donc pas être encapsulées dans une classe d'objets. Du fait qu'il s'agit de variables globales, on aurait pu se contenter de les modifier et de les consulter dans les différentes classes de services. Cependant, les informations de session représentent bien un concept indépendant. L'isolation des méthodes de vérification de session dans un fichier indépendant permet à celle-ci d'être altérée aisément.

A l'avenir, on peut choisir de mettre un délai d'expiration sur les sessions d'utilisateurs, ou d'incorporer de nouveaux niveaux d'utilisateurs. Pour de tels changements, seule l'implémentation des méthodes contenues dans ce fichier auront à être modifiées. Par ailleurs, cette «librairie» pourra être réutilisée par une autre interface (celle d'acquisition de données, par exemple) pour avoir aussitôt un système de session commun aux deux interfaces. Par contre, l'authentification d'un utilisateur ne passe pas par ce fichier qui n'accède pas à la base de données. En effet, les méthodes se contentent de créer et modifier la session. Ce choix permet de rendre cet ensemble de méthodes réutilisable quelle que soit le mode d'authentification (utilisateurs dans un fichier, dans une base de données, avec ou sans mot de passe...) pour définir les sessions. En effet, dans notre architecture, la classe `UtilisateurPhp` propose une méthode d'authentification qui crée la session de l'utilisateur, au travers de la méthode importée avec *Session.php*, si les informations de l'utilisateur correspondent à un enregistrement de la table `utilisateur`. Les autres classes importent également le fichier pour vérifier que l'utilisateur est bien administrateur ou correctement authentifié dépendant du cas de figure.

Enfin, PHP est un langage beaucoup moins exigeant qu'ActionScript au niveau du typage de ses variables. En effet, la déclaration des méthodes n'impose pas de type de variable particulier en entrée, et ne stipule pas la valeur de retour de l'objet retourné. C'est au programmeur de s'assurer que sa méthode renvoie bien ce qu'il souhaite ainsi que de la validité des arguments qu'on lui transmet. A la création de la classe ActionScript homologue, pour l'application cliente, les services vont prendre en paramètre et renvoyer des objets au sens large (du type ActionScript Object). La méthode *addEntry($rubrique,$contenu,$admin)* de la classe `AidePhp` deviendra simplement *addEntry(rubrique :Object,contenu :Object,admin :Object*. Néanmoins, l'appel distant fonctionne également si l'on précise le type des attributs (*addEntry(rubrique :String,contenu :String,admin :String*) ce qui, du point de vue du développeur, l'empêche de déclarer des opérations avec des paramètres mal choisis et donc d'effectuer des erreurs de codage. La valeur de retour d'un appel de méthode est, elle aussi, imprécisément définie en tant que «Object» puisque le langage PHP n'offre aucune garantie sur le type de la valeur de retour.

Afin de contourner ce problème on peut vérifier que la fonction renvoie bien un type plus précis («cast» la valeur de retour comme étant d'un certain type). Moyennant réussite de cette vérification la variable est alors typée plus précisément et nous pourrons la manipuler avec les méthodes qui lui sont propre.

4.3.3 Conclusion

A la suite de cette modélisation détaillée, l'implémentation est bien plus aisée. La modélisation initiale a tout de même évolué au fur et à mesure du développement pour prendre en compte des éléments d'implémentation qui n'avaient pas été envisagés initialement. Néanmoins, dans l'ensemble, l'implémentation finale est le reflet de la modélisation de départ. Les diagrammes qui ont été repris dans cette section tiennent compte des modifications apportées durant le développement. Ils constituent donc une documentation précise du logiciel qui devrait permettre à une tierce personne de reprendre le projet sans avoir la difficulté que nous avons rencontrée avec Zope.

4.4 Le produit fini

4.4.1 Installation

4.4.1.1 Configuation minimale sur le serveur et chez le client

L'extension PDO est automatiquement inclue dans PHP à partir de la version 5.1. Il s'agit de la seule contrainte de configuration requise ; les composants de Zend inclus dans le dossier suffisent à la bonne exécution du framework.

Chez le client, l'utilisation de Flex 4 réclame la version 10 de FlashPlayer qui est fort bien répendue actuellement.

4.4.1.2 Configuration de l'application

Deux fichiers de configuration permettent de paramétrer l'installation :

amf_config.ini : les paramètres de la configuration AMF y sont définis. Deux paramètres doivent être adaptés : le répertoire racine du web sur l'hôte et le répertoire contenant les fichiers PHP de services (servPHP). Le fichier est dans le répertoire principal de l'application : interfaceAC.

db.conf : les paramètres de connexion à la base de données y sont définis. Il est dans le dossier *servPHP*. Il est important que les différents éléments de configuration restent sur des lignes indépendantes.

Des modifications doivent être apportées à la base de données [20] : 3 tables (aide, question et utilisateur) et 2 fonctions (disable_state_changes et enable_state_changes). Seule la table utilisateur d'origine est modifiée, toutes les autres modifications se greffent sur le système en place et n'altèrent pas son fonctionnement pour les autres interfaces.

4.4.2 Modifications apportées à la base de données

La base de données comprend dorénavant deux nouvelles tables et deux nouvelles fonctions. La première table permet d'enregistrer les questions courantes de l'utilisateur afin qu'il puisse reproduire la même recherche.

4.4.2.1 Modifications apportées aux tables

A. A. Description des utilisateurs

Deux champs ont été ajoutés à la table `UTILISATEUR` : un champ contenant le mot de passe (*mdpasse*) et un champ contenant le niveau d'administration de l'utilisateur : *admin*. Le niveau d'administrateur est actuellement 1 tandis que l'utilisateur n'ayant que des droits consultatifs est de niveau 0 mais c'est une

[20] Le code source de ces modifications est inclu en annexe.

convention qui peut être facilement adaptable notamment si l'on souhaite à l'avenir introduire des utilisateurs de niveau intermédiaire. Le mot de passe est crypté en SHA1 par l'interface. Les deux nouveaux champs sont facultatifs.

B. B. Description de l'aide

La table `AIDE` vient d'être introduite. Elle permet d'enregistrer les rubriques d'aide à l'utilisateur de l'interface. Chaque enregistrement comporte cinq colonnes :

id : identifiant des différents enregistrements de la table.

admin : l'aide n'est pas restreinte aux utilisateurs administrateurs. Certaines rubriques s'adresseront donc à l'utilisateur administrateur tandis que d'autres pourront concerner l'utilisateur commun. Le champ *admin* permet d'adapter le contenu de l'aide à l'utilisateur qui la consulte.

rubrique : tout article disponible dans l'aide est associé à une rubrique. Un ensemble d'articles peuvent être rattachés à la même rubrique. Sont considérés comme appartenant à la même rubrique deux articles possèdant le même intitulé de rubrique exactement.

titre : ce champ permet d'enregistrer le titre des différents articles.

description : le champ *description* contient le contenu informationnel de l'aide proprement dite.

C. Description des questions

Une table `QUESTION` permet d'enregistrer l'ensemble des informations relatives aux requêtes pré-enregistrées. La table ne contient que quatre colonnes. Elle permet de reconstituer des requêtes simples basées sur les formulaires de recherche qui sont toujours associées à une table spécifique.

id : identifiant des différentes questions

table : indique la table sur laquelle porte la requête

contraintes : permet l'enregistrement des contraintes de la requête. Elles sont enregistrées sous la forme d'une chaîne de caractères comprenant où chaque contrainte est séparée de la suivante par un 'point-virgule'. La représentation d'une contrainte est dépendante de l'interface. En effet, elle donne l'indice du champ dans le formulaire auquel se rapporte l'information et donne la valeur avec laquelle il doit être prérempli. Cette dépendance face à l'interface est un choix afin de ne pas donner, au travers d'un formalisme proche de la base de données, d'indications sur la structure et les dénominations employées dans celle-ci [21].

description : la description d'une question contient la phrase qui va permettre à l'utilisateur d'identifier la recherche qu'effectue la question.

4.4.2.2 Fonctions de la base de données

Deux fonctions complètent la base de données actuelle. Elles permettent de contourner le système de double validation mis en place au travers des triggers. La première fonction (*disable_state_changes*) désactive les triggers contrôlant les états de la mise à jour, la suppression et l'insertion au sein de la base de données. La deuxième fonction (*enable_state_changes*) rétablit les différents triggers suspendus par la première.

L'aboutissement du développement n'est pas uniquement l'occasion de découvrir une nouvelle interface du système mais également l'occasion de s'interroger sur le respect des exigences établies par l'analyse du chapitre 3 et de s'assurer qu'on a bien évité de tomber dans les travers du premier développement. Les arborescences de tâches définies dans le courant de l'analyse des exigences sont repectés dans l'interface finale. Les maquettes initiales ont évolué légèrement mais les grandes lignes qui avaient été déterminées ont été respectées. Nous avons pris des captures d'écran sous Internet Explorer et sous Firefox afin de se convaincre de la représentation équivalente de l'interface d'un navigateur à l'autre. Les différentes fonctionnalités de l'interface ont été testées sous ces deux navigateurs et elles fonctionnent de la même manière[22].

[21] En effet, les fichiers .swf étant facilement décompilables, il est important de limiter au maximum l'ensemble des indications sur le fonctionnement de la base de données et des classes php auprès dans le développement en Flex.

[22] Une machine virtuelle accompagne ce mémoire. Elle comprend l'interface installée sur un système d'exploitation Ubuntu qui peut ainsi être testée. Son installation est décrite en annexe.

4.4.3 Éléments accessibles à tout utilisateur

L'analyse des exigences a mis en évidence une nouvelle attente : la définition de deux niveaux d'utilisateurs. Un premier ensemble de fonctionnalités sont communes à tous les utilisateurs. Elles comprennent l'opération intiale d'authentification, les consultations de la base de données, et l'accès à une section aide.

4.4.3.1 Accueil

A la suite d'un chargement initial[23], l'utilisateur aboutit sur la fenêtre de la figure 4.17. C'est une simple fenêtre d'accueil. Une bonne complétion des champs nous voit aboutir sur une interface présentant quatre onglets. Selon qu'il est admin ou non, l'utilisateur peut ou non accéder au second et quatrième onglet : l'insertion de données et la gestion des utilisateurs respectivement. L'interface est volontairement sobre car nous recherchions une apparence proche de l'interface d'acquisition de données existante.

4.4.3.2 La consultation des données

L'onglet présenté par défaut est l'onglet de consultation. Il comporte trois sous menus : Questions, Recherche et Naviguer. A l'ouverture de l'application on aboutit par défaut dans la section «Question» qui présente la liste des recherches pré-enregistrées par l'utilisateur. La possibilité d'enregistrer des questions est une exigence qui est apparue en cours de développement du fait de la constatation qu'un utilisateur aura tendance à effectuer régulièrement les mêmes recherches. La sélection d'une question ouvre la section «Recherche» (voir figure 4.18) où l'un des formulaires est prérempli avec les contraintes définies par la recherche. La section «Recherche» propose cinq recherches directes. A la sélection de l'une d'entre-elles le formulaire offrant les champs de recherche qui lui sont spécifiques apparaît à droite. L'utilisateur peut le compléter afin d'éffectuer une recherche.

S'il le souhaite l'utilisateur peut choisir d'enregistrer cette recherche. Un formulaire lui permettant d'en indiquer la description apparaît alors (figure 4.19). A la validation du formulaire, l'utilisateur reçoit une indication qu'elle a bien été enregistrée mais reste dans la section «Recherche».

Lorsque l'utilisateur a complété son formulaire et qu'il sélectionne *Rechercher*, il est redirigé dans la section «Naviguer» vide jusqu'alors. Elle propose les résultats de la recherche et permet de naviguer au sein de ceux-ci. L'ensemble des résultats se présente comme une liste de liens cliquables (voir la figure 4.20) présentant un titre pour le type d'entité recherché. En cliquant sur un lien présentant, en l'occurrence, un matériau, on déploie sa case dans laquelle figurent toutes ses caractéristiques propres (figure 4.21). En plus de ces caractéristiques propres, la case d'un type d'entité peut présenter un ensemble de liens vers des caractéristiques étendues. Elles permettent de déployer des informations annexes : pour un matériau, les propriétés qui le qualifient par exemple. En sélectionnant de déployer ces informations une liste de nouveaux liens cliquables présentant maintenant ces caractéristiques se déplie. L'ensemble des relations que l'on peut parcourir depuis un type d'entité est représenté graphiquement à la figure 4.10). A la figure 4.22 nous venons de choisir de déployer l'ensemble des quantifications auxquelles est lié ce matériau et de sélectionner l'unique quantification dont il fait l'objet.

4.4.3.3 L'aide

Une section d'aide est accessible depuis les deux interfaces. Elle présente deux panneaux. Le premier contient un ensemble de rubriques cliquables. Lorsque l'on clique sur une rubrique son contenu apparaît dans le second panneau (droite). Ce développement ne faisait pas partie des exigences initiales mais est apparu, dans le courant du développement, comme une opération nécessaire. En effet, le fait d'ouvrir l'interface à un nouveau type d'utilisateur, moins averti, qui ne connait pas l'interface et doit apprendre comment y faire des recherches rendait cette fonctionnalité initialement triviale beaucoup plus importante.

4.4.4 Interfaces spécifiques à l'administrateur

Comme convenu dans l'analyse des exigences, un ensemble d'opérations n'est accessible qu'à un utilisateur de niveau supérieur. Celui-ci peut procéder aux modifications dans la base de données en plus des

[23]Ce temps de chargement est propre à toutes les applications Flex et lié au fait qu'après le téléchargement elles sont compilées chez le client.

consultations. C'est important puisque ça ne comprend pas que les insertions mais également les éditions et les suppressions des différents éléments.

Afin de procéder de manière structurée et cohérente, nous avons maintenu l'ordre de priorité défini dans l'analyse des exigences. Il reprenait un ensemble d'élements pour lesquels il était impératif de développer des formulaires d'encodage. Les éléments insérables (solutions, matériaux, interactions, procédés, ...) sont également éditables et supprimables. Un développement futur devrait envisager la rapide complétion de l'application en rendant tous les types d'entités insérables, éditables et supprimables au travers de l'interface.

4.4.4.1 Fonctionnalités supplémentaires sur les interfaces communes

L'utilisateur administrateur accède forcément à un ensemble de fonctionnalités qui lui sont spécifiques.

Dans l'interface de consultation, il lui est possible d'éditer et de supprimer des questions. L'édition se fait au travers d'un formulaire qui apparaît lorsque l'on sélectionne l'icône d'édition (voir figure 4.23). La suppression fonctionne de la même manière mais fait apparaître une fenêtre de confirmation. La même démarche est possible avec les différents composants déployés dans la recherche navigationnelle. Un petit bémol mérite d'être souligné : les propriétés héritées de parents (voir la section choix d'implémentation) ne sont pas éditables au travers de cette interface dont la portée se limite à une structure plane de données. Finalement, les rubriques dans l'aide peuvent être crées, supprimées et éditées également.

L'interface de l'administrateur lui permet également d'enregistrer les recherches (section «Recherche» de l'onglet «Consulter») sous la forme de questions types proposées dans la section «Questions» de l'onglet de recherche. Dans un formulaire de recherche, lorsqu'une mention est laissée vide, l'application estime qu'elle n'intervient pas. Ainsi, laisser la case couche décochée ne signifie pas que l'on recherche les matériaux qui ne sont pas des couches mais que le champ couche n'intervient pas en tant que critère de recherche.

4.4.4.2 Insertions

L'onglet le plus intéressant pour l'administrateur est certainement l'onglet d'insertion. Il va permettre d'insérer, comme convenu lors de l'analyse des exigences, des matériaux, des interactions, des procédés et des incompatibilités. L'insertion implique la complétion de formulaires proches de ceux qui étaient proposés dans l'interface précédente. Les différents champs qui permettaient de retrouver une feuille des familles ont été abandonnés. Ces systèmes permettaient de retrouver l'enregistrement souhaité au travers de listes raccourcies. On sélectionnait la racine dans une liste de familles et l'ensemble des enregistrements potentiels se réduisait au fur et à mesure qu'on se rapprochait de la base de la hiérarchie. Dans l'interface actuelle, il a été convenu d'utiliser les identifiants numériques pour tous les champs référençant d'autres enregistrements. Il fallait néanmoins proposer une solution pour trouver l'enregistrement à référencer si on n'en connaissait pas l'identifiant sans passer par la recherche. La solution proposée par l'interface présente un panneau latéral (comme le panneau *propriété* à la figure 4.24) avec la liste des noms des différents enregistrements référençables par un champ. Pour faire apparaître le panneau de recherche rapide une icône de recherche est juxtaposée aux champs qui requièrent l'identifiant numérique d'un enregistrement en base de données. Lorsque l'on clique sur un élément de la liste dans le panneau, il préremplit le champ associé. Le panneau peut rester ouvert aussi longtemps que l'utilisateur souhaite en consulter les données et reste visible malgré les changements d'onglets[24].

Tous les champs des différents formulaires s'assurent du respect de toutes les contraintes d'intégrité de la base de données avant de sauvegarder l'enregistrement (localement[25]) et de l'insérer ensuite dans la base de données. Cette vérification se présente comme à la figure 4.25. Lorsque l'utilisateur confirme qu'il souhaite valider son encodage, une vérification est faite de la validité des données encodées. Si elles sont vérifiées alors l'enregistrement est sauvegardé localement. Sinon, il présente un message d'erreur et encadre en rouge tous les champs concernés.

Les enregistrements sauvegardés sont présentés dans la grille de la partie de l'insertion concernée. Ils peuvent être édités, supprimés (sans confirmation puisqu'il s'agit d'un enregistrement local) et même copiés. Les différentes encodages peuvent enfin être insérés définitivement dans la base de données à l'aide du bouton *Insérer les encodages*.

[24] Attention qu'il reste toutefois lié au même champ tant que l'on ne clique pas sur une autre icône de recherche rapide

[25] L'enregistrement est inséré dans un tableau en mémoire sur la machine. Le serveur n'intervient pas dans ces opérations.

4.4.4.3 Gestion des utilisateurs

L'onglet «Gestion utilisateurs» (voir figure 4.26) permet de créer, d'éditer (figure 4.27) et de supprimer les utilisateurs. L'interface est simple et ne présente pas de difficulté majeure.

4.5 Tests

Le début de la phase de développement n'a pas permis d'effectuer des tests en parallèle dans la mesure où il fallait construire l'architecture et les «fondations» de l'application. En revanche, dés que des ensembles de fonctionnalités sont arrivées au bout de leur développement, une installation a été faite chez le client pour lui permettre de faire part au fur et à mesure de ses remarques et des problèmes de l'interface.

Dans le cadre d'une interface la majorité des tests concernent en définitive des tests d'utilisation de l'interface. Certaines parties du développement présentaient tout de même quelques difficultés logiques telles que l'écrasement des relations familiales. Afin de vérifier la validité de ces résultats nous avons procédé par études de cas : une comparaison des résultats avec ceux que nous savions devoir obtenir. Ce procédé ne garantit pas la validité de tous les résultats mais effectué dans le cadre de tous les cas limites augmente les probabilités que la fonction soit valable en toute circonstances.

Vu la complexité du domaine d'application, la seule vérification de la validité des résultats par le programmeur n'était pas suffisante. Dés que des ensembles de fonctionnalités sont arrivées au bout de leur développement, une machine virtuelle comprenant l'ensemble du système Expesurf doté de sa nouvelle interface a été installée chez le client. L'application a donc été testée par les utilisateurs afin de s'assurer de son bon fonctionnement et qu'elle répondait bien aux attentes formulées. Une utilisation prolongée de l'application permettra certainement de mettre en évidence d'autres dysfonctionnements et ou des exigences qui n'apparaissaient pas jusque là mais les corrections qu'il y avait moyen d'identifier et d'effectuer dans les délais ont été apportées.

Ainsi, les formulaires de recherche ont été complétés de listes de recherche rapide. Les recherches portent dorénavant sur tous les enregistrements de la base de données plutôt que seulement les enregistrements enfants lorsque ceux-ci étaient disponibles. Certaines fiches de résultats ont été complétées de champs manquants.

Un vérification a été ajoutée lorsque l'on quitte l'application pour s'assurer que l'utilisateur ne quitte pas la page par mégarde. En effet, l'application Flash est intégrée à une page HTML. De ce fait, le bouton «page précédente» du navigateur ne permet pas de s'orienter au sein de l'application Flash mais entraîne un retour à la consultation précédant le démarrage de l'application. Pour un utilisateur novice, habitué à se servir régulièrement de cet outil de navigation, cela peut sembler déconcertant et il était donc important de l'avertir lorsqu'il quittait la page.

D'autres remarques liées à la phase de test n'ont en revanche pas pu être corrigées ou mises en place. Elles concernaient principalement le développement de nouvelles fonctionnalités et constituent des premières pistes pour des développements futurs de l'application.

Il serait important de noter que la phase de test est insuffisante à l'heure actuelle. Le contexte d'utilisation actuel du logiciel n'est pas, à priori, le contexte d'utilisation le plus courant. Notre client/utilisateur est un expert qui connait le système expert et sa base de données. Il a, par ailleurs, déjà manipulé des interfaces similaires et a des attentes personnelles qui ne reflètent pas nécessairement celles des clients potentiels. Il serait intéressant pour des développements futurs d'envisager de pouvoir soumettre l'application à une communauté d'utilisateurs pour récolter une critique plus complète et plus diversifiée de l'interface. Grâce à une telle communauté on pourrait soumettre les utilisateurs à des tests beaucoup plus formalisés en passant par des questionnaires ou en leur faisant effectuer des tâches élémentaires et en évaluant la difficulté de la tâche et le temps nécessaire pour l'effectuer. Si le système expert base une grande partie de sa réussite sur le contenu de sa base de données et sur la qualité du moteur d'inférence, il serait naïf de négliger l'impact d'une interface inefficace sur la popularité du système expert dans son ensemble. C'est le confort d'utilisation de l'interface qui va permettre de mettre en valeur ou, à l'inverse, de discréditer l'ensemble du système expert avant même que la qualité des résultats soient pris en compte.

4.6 Conclusions

Si les résultats de l'analyse des exigences présentent fidèlement les attentes et les besoins du client, le développement devrait être promis à une belle longévité. En effet, l'architecture de l'application devrait la rendre facilement accessible pour tout développeur qui chercherait à s'approprier le code pour des développements futurs. La modularité de l'application la rend facilement adaptable également à de nouvelles configurations. Que l'on veuille remodeler l'agencement de l'interface, les considération stylistiques de celle-ci, ou revoir la présentation des résultats, chaque adaptation ne touchera qu'à la portion de code qui la concerne spécifiquement sans avoir à modifier le reste de l'application.

Ensuite, le choix d'employer une technologie utilisée largement mais récente par la même occasion devrait garantir la possibilité de compléter indéfiniment l'interface de nouveaux composants. La communauté développant en Flex est vaste. Cette communauté offre un service de soutien au programmeur mais développe également de nouveaux composants puissants qui offrent, en plus des outils initiaux de flex, de nouvelles possibilités de développements allant d'amélioration des composants d'impression, d'exportation de données en toutes sortes de formats, d'outils de pagination des résultats, ou encore d'outils facilitant la représentation de données complexes.

En définitive, ce développement n'est pas un développement final. Il reste la possibilité de nombreuses améliorations d'une part, et un travail incomplet sur certaines fonctionnalités telles que les insertions, éditions et suppressions qui ne permettent pas de travailler sur toutes les tables de la base de données. En revanche, le travail accompli représente une bonne base capable de remplacer l'interface d'acquisition de connaissances actuelle et présentant une recherche capable de mettre en évidence des incohérences dans la base de données. De plus, la nouvelle interface est dotée d'une architecture réfléchie et modulaire qui facilitera le reprise du projet par un développeur quelconque par la suite. Le choix technologique permettra certainement de nombreux développement futurs et une longévité certaine à l'application.

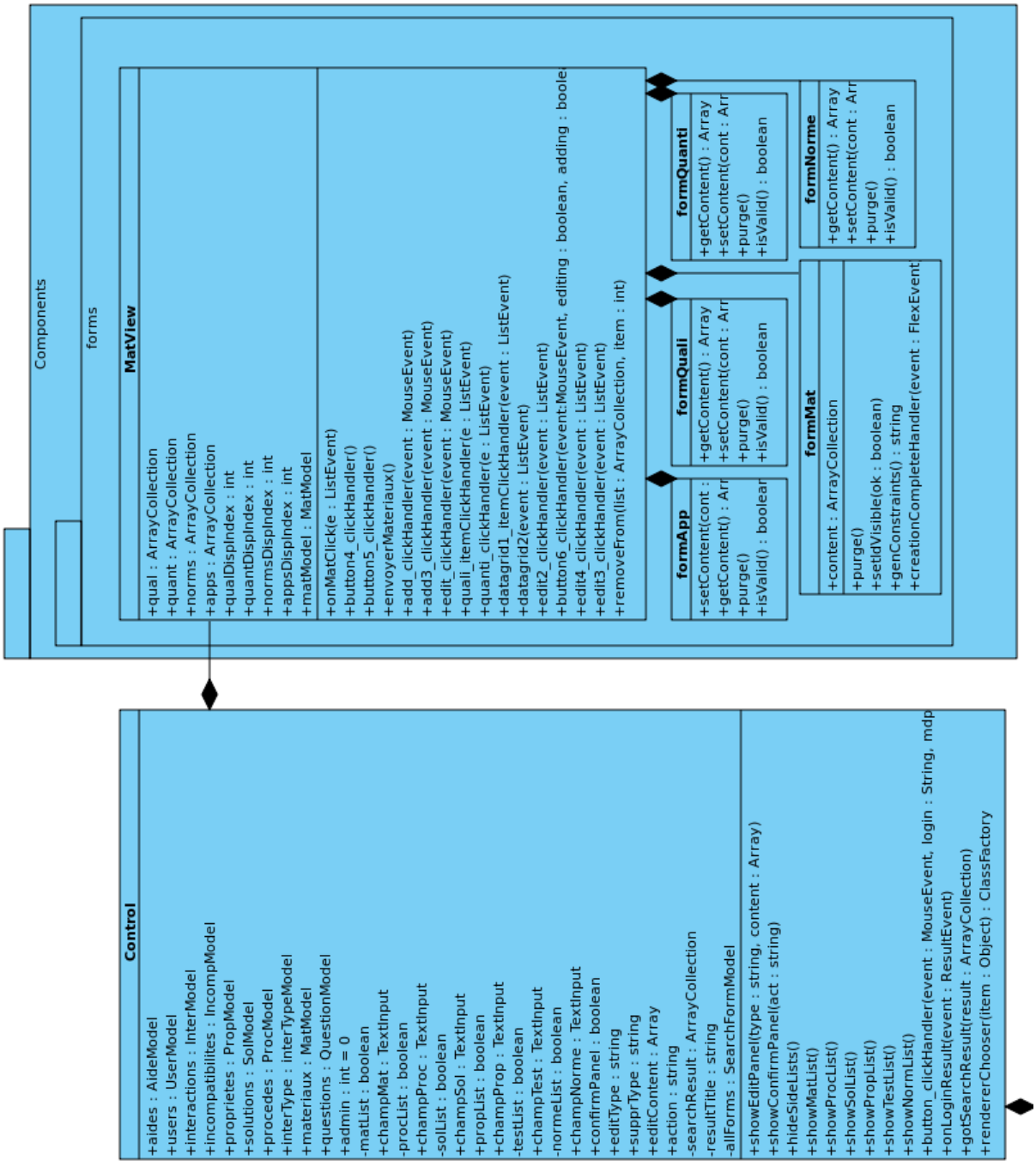


FIG. 4.11 – Diagramme de classe Insertion d'un matériau

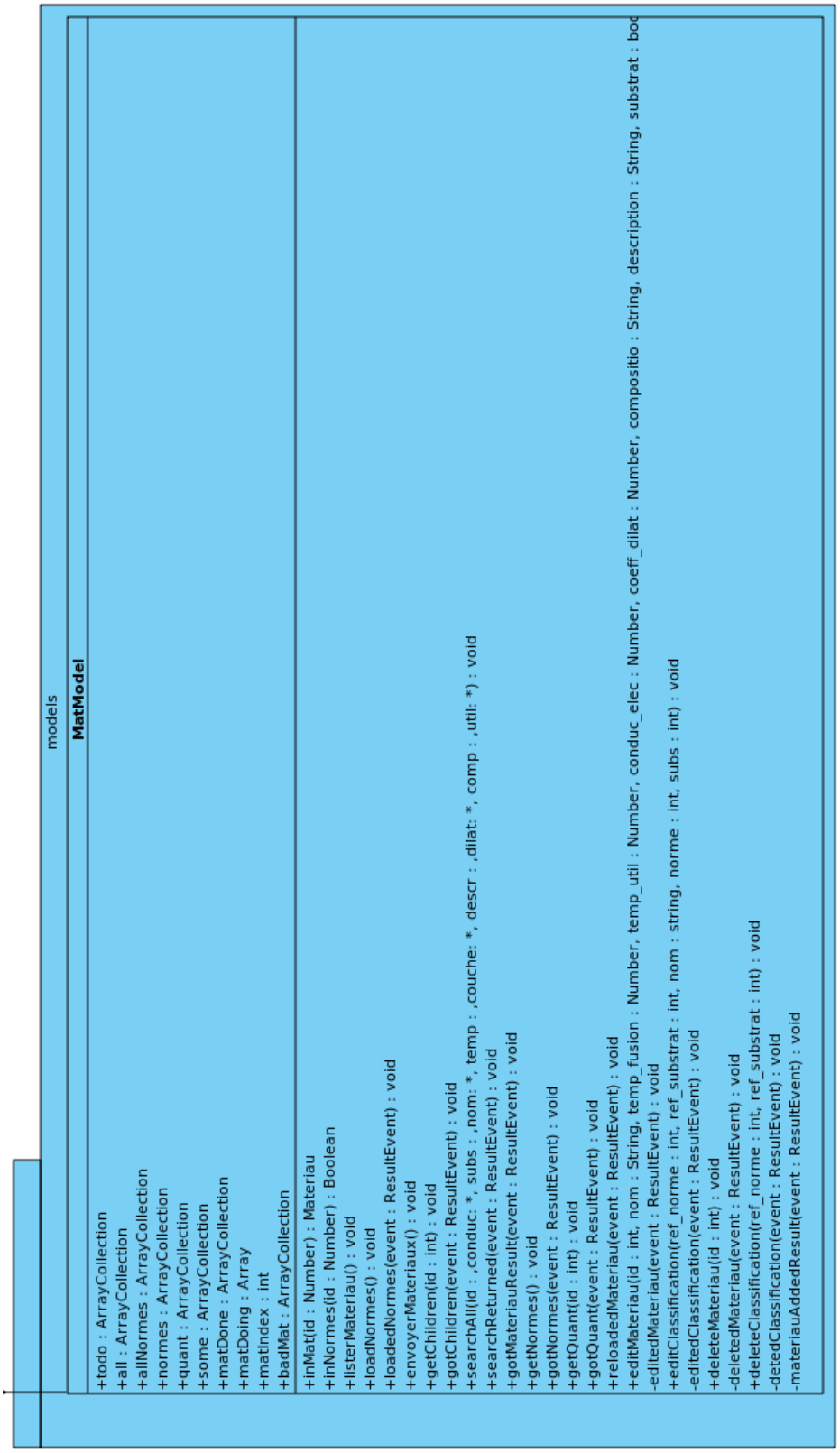


FIG. 4.12 – Diagramme de classe Insertion d'un matériau

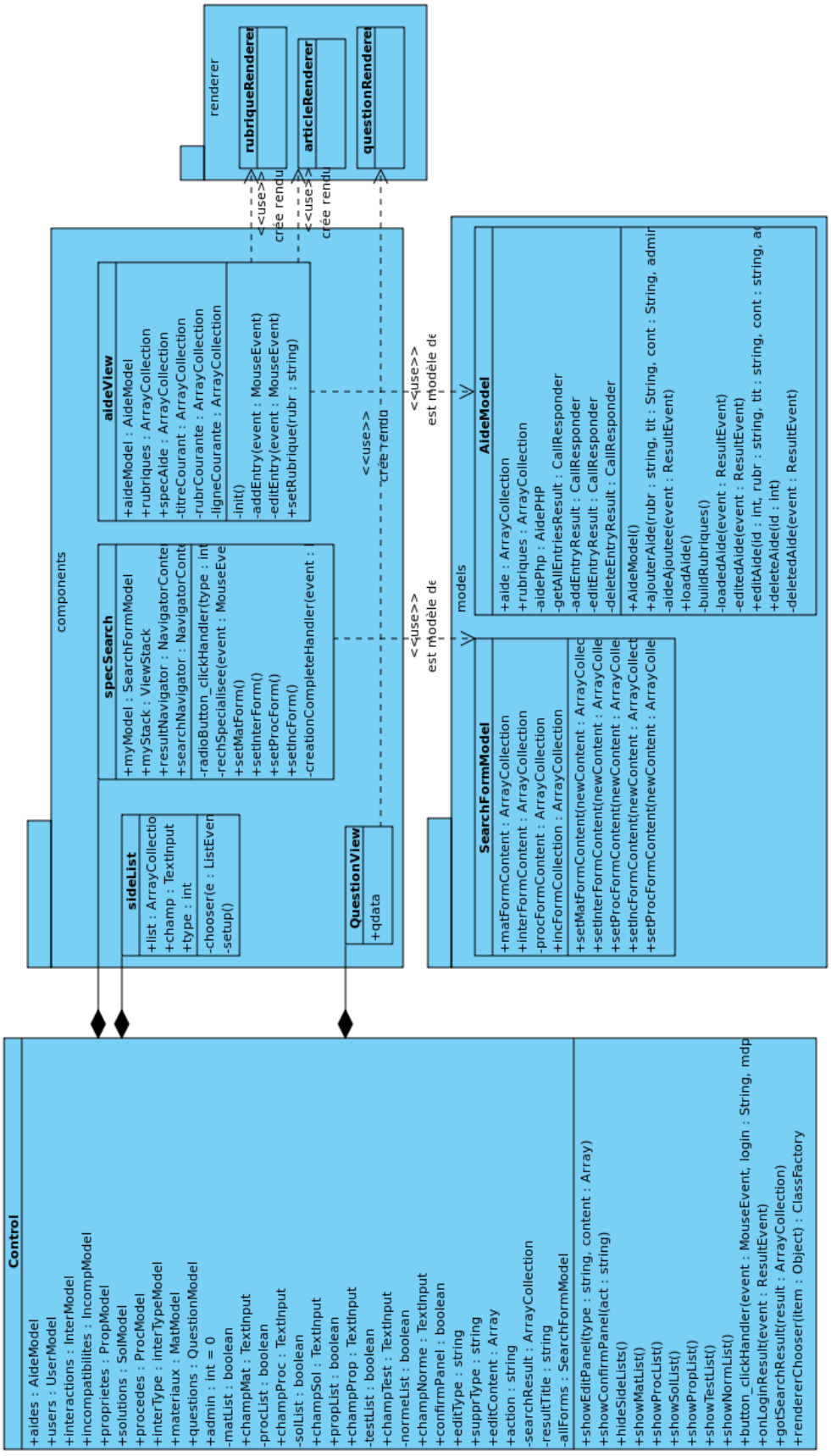


FIG. 4.13 – Diagramme de classe de l'aide et de la recherche

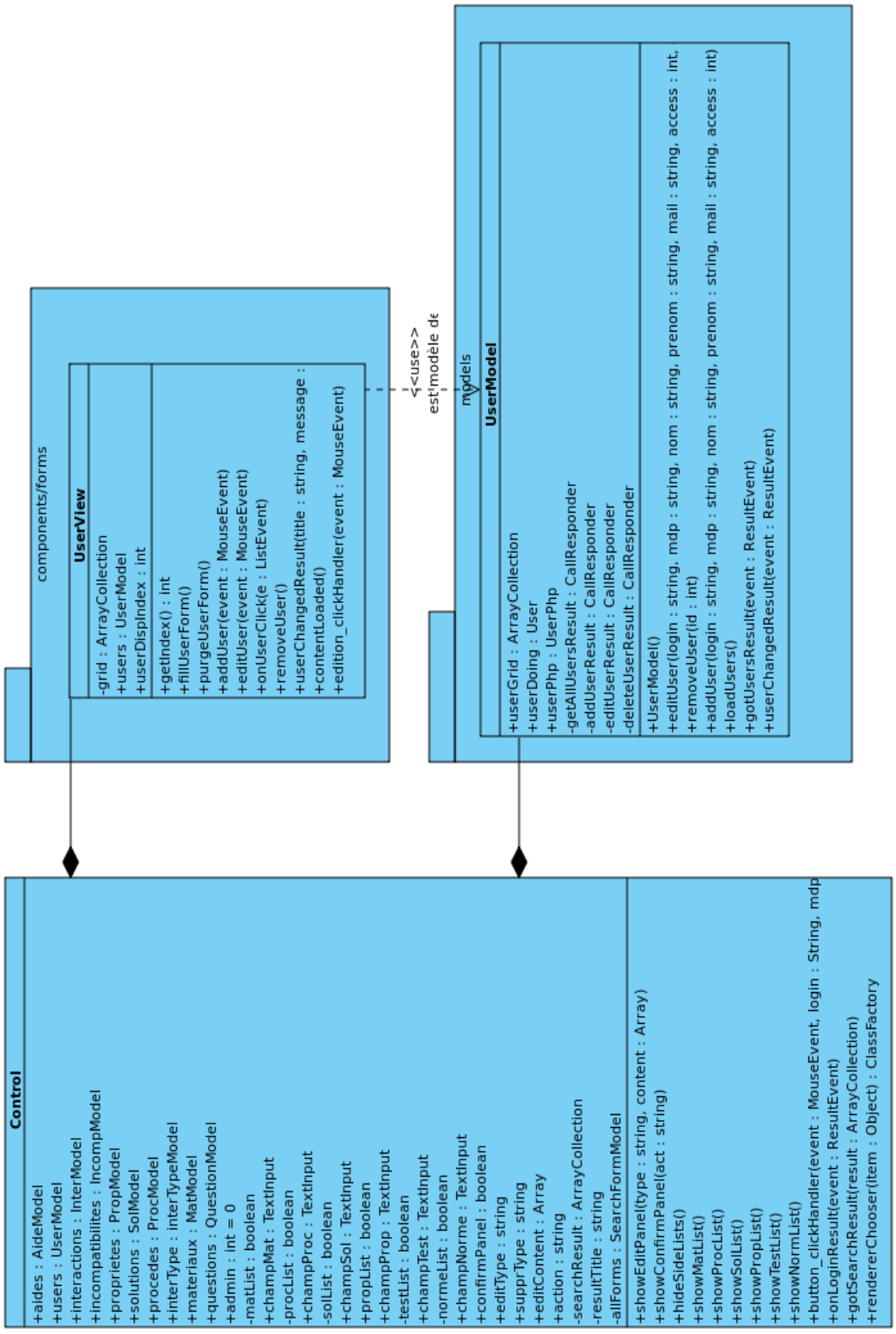


FIG. 4.14 – Diagramme de classes de la gestion des utilisateurs

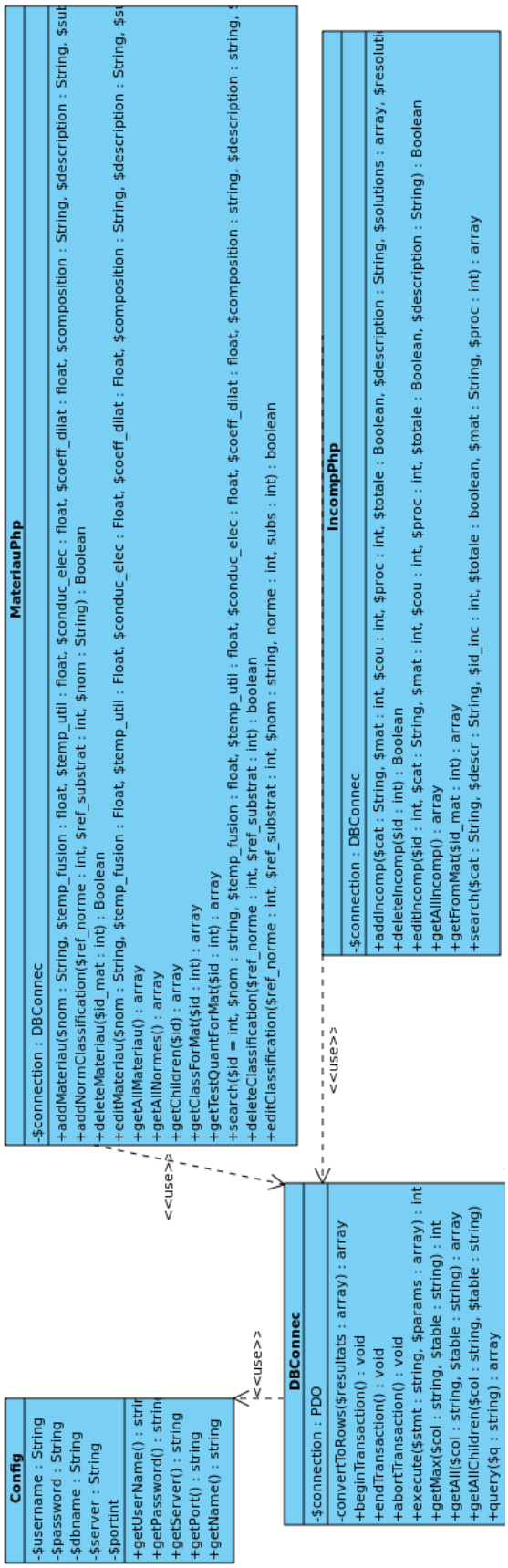


FIG. 4.15 – Diagramme de classes des services publiés par le serveur

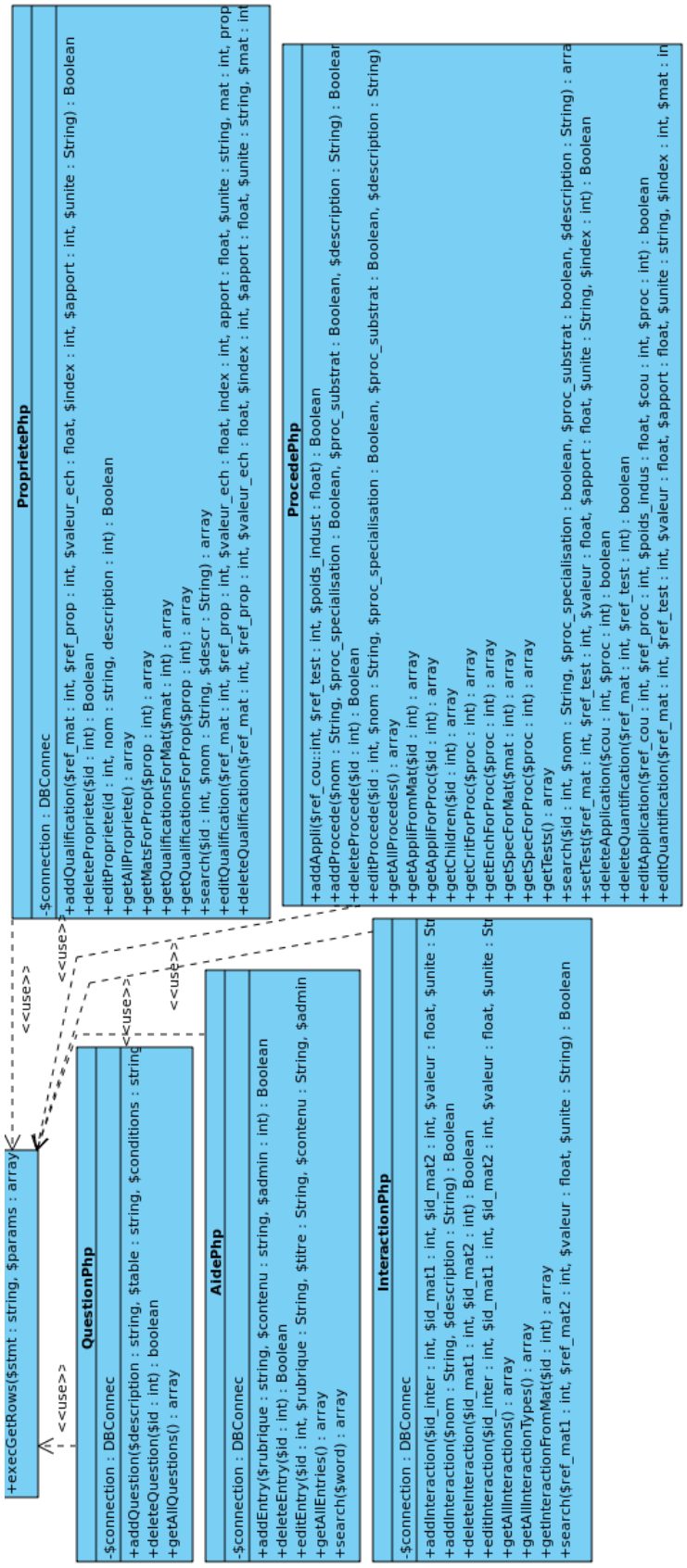


FIG. 4.16 – Diagramme de classes des services publiés par le serveur

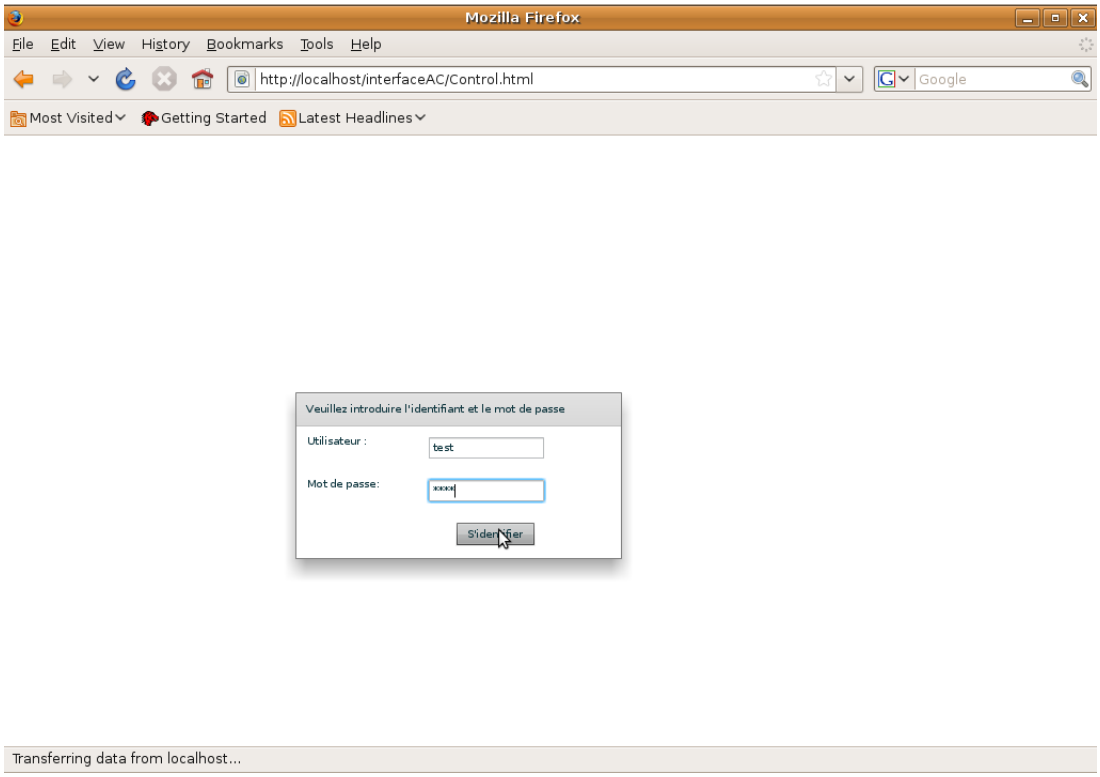


FIG. 4.17 – Accueil

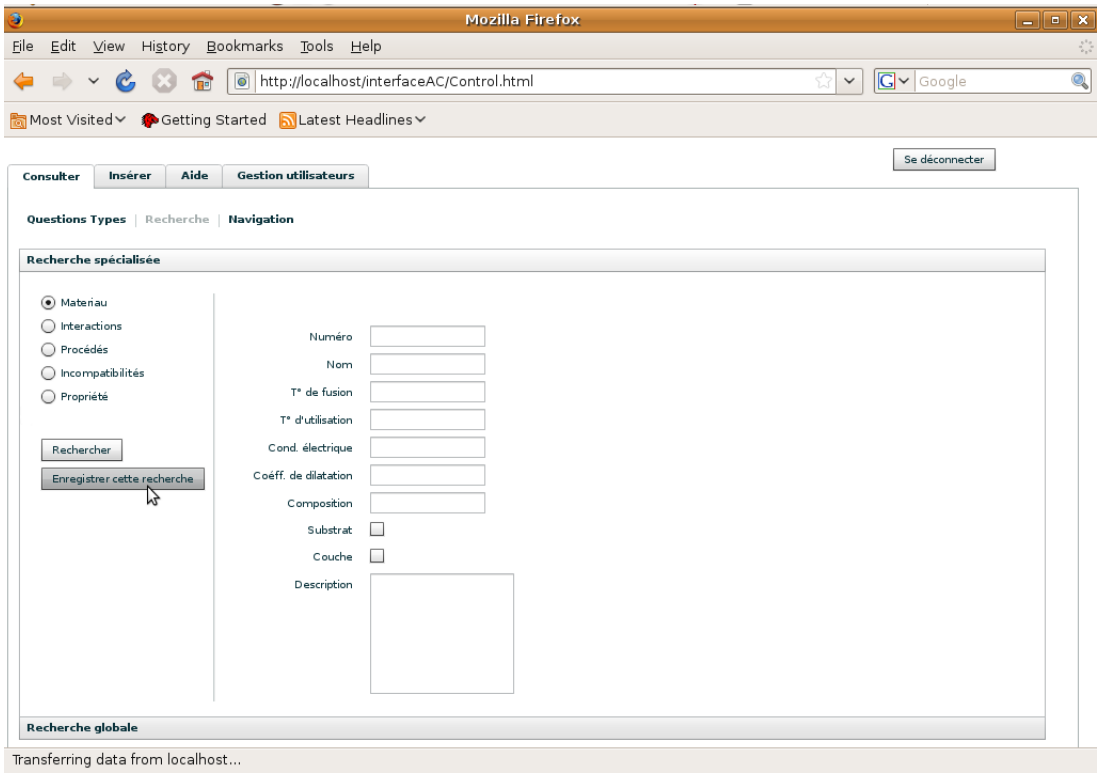


FIG. 4.18 – Recherche

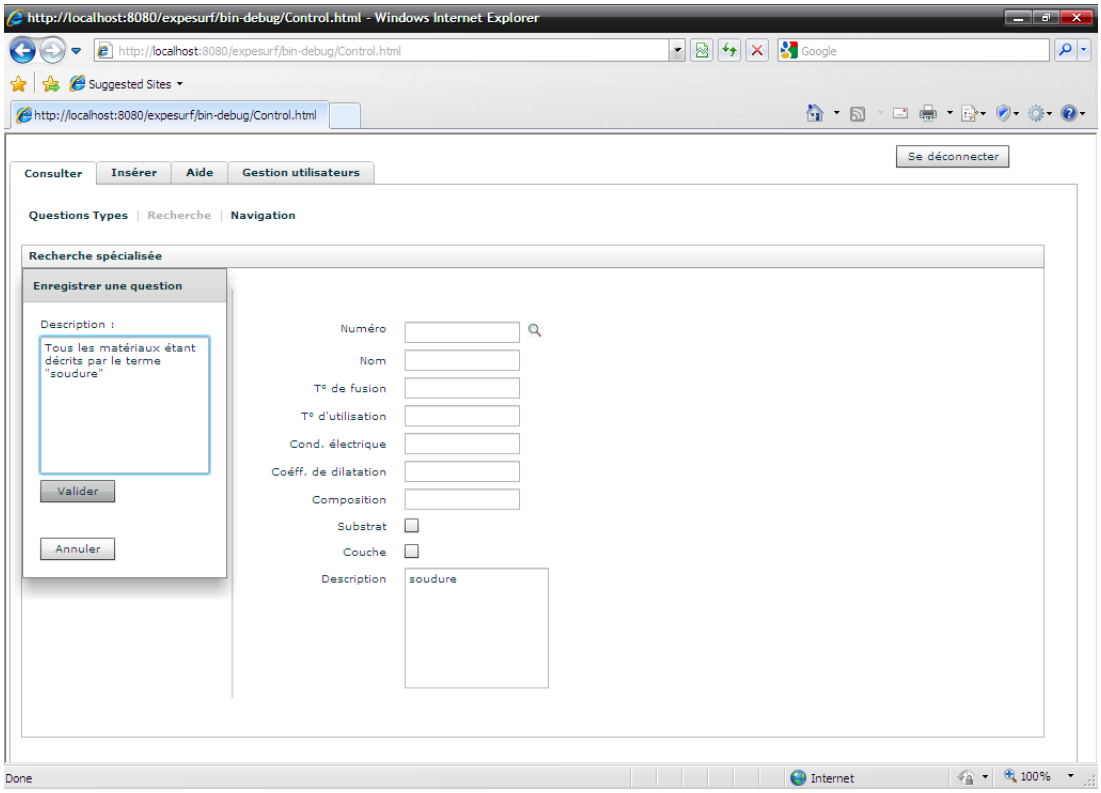


FIG. 4.19 – Enregistrer une question

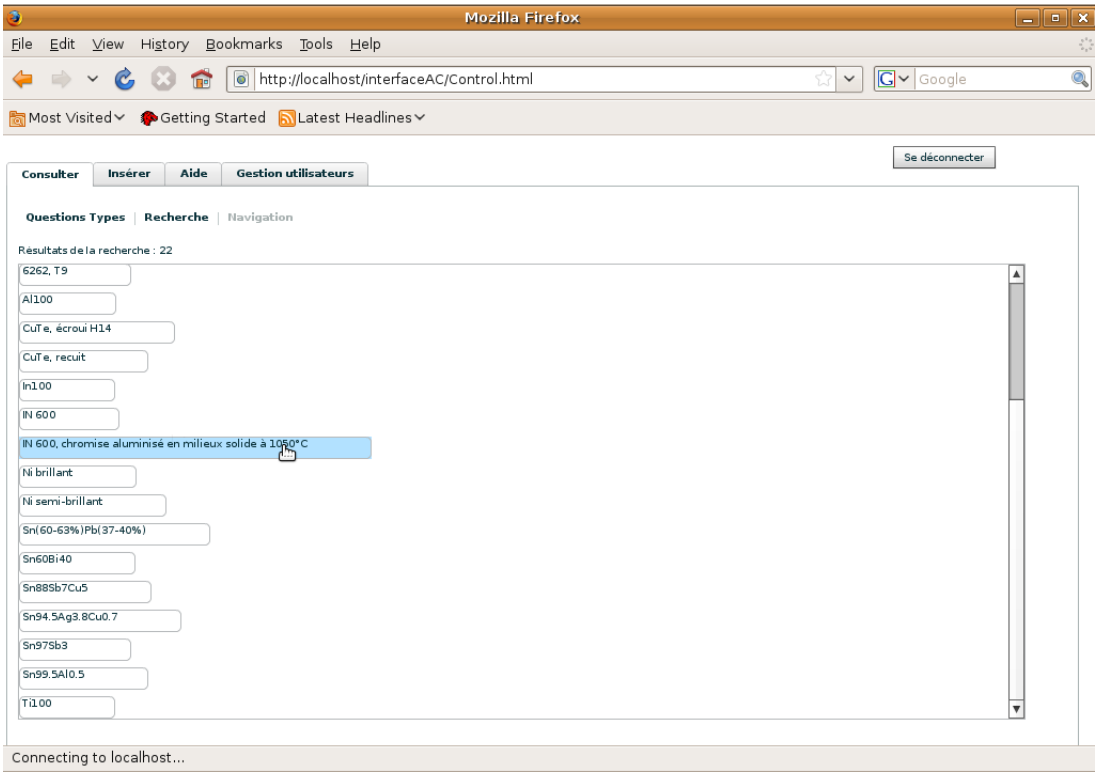


FIG. 4.20 – Résultats de recherche d'un matériau

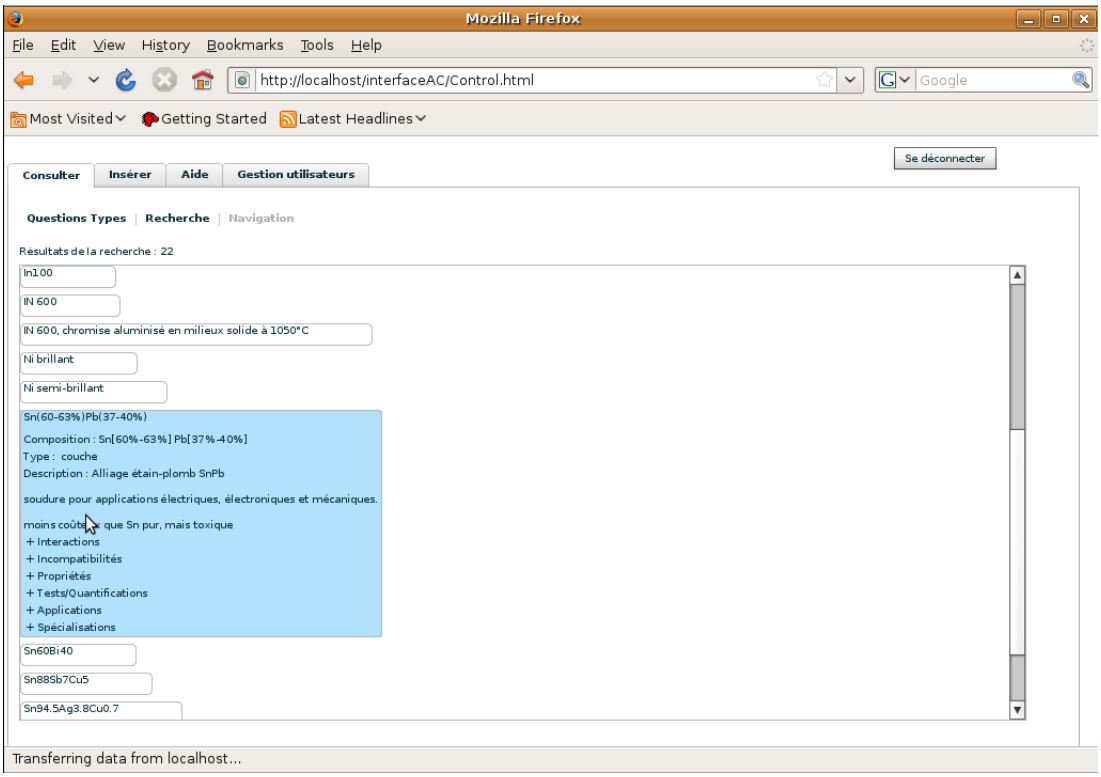


FIG. 4.21 – Matériau déployé

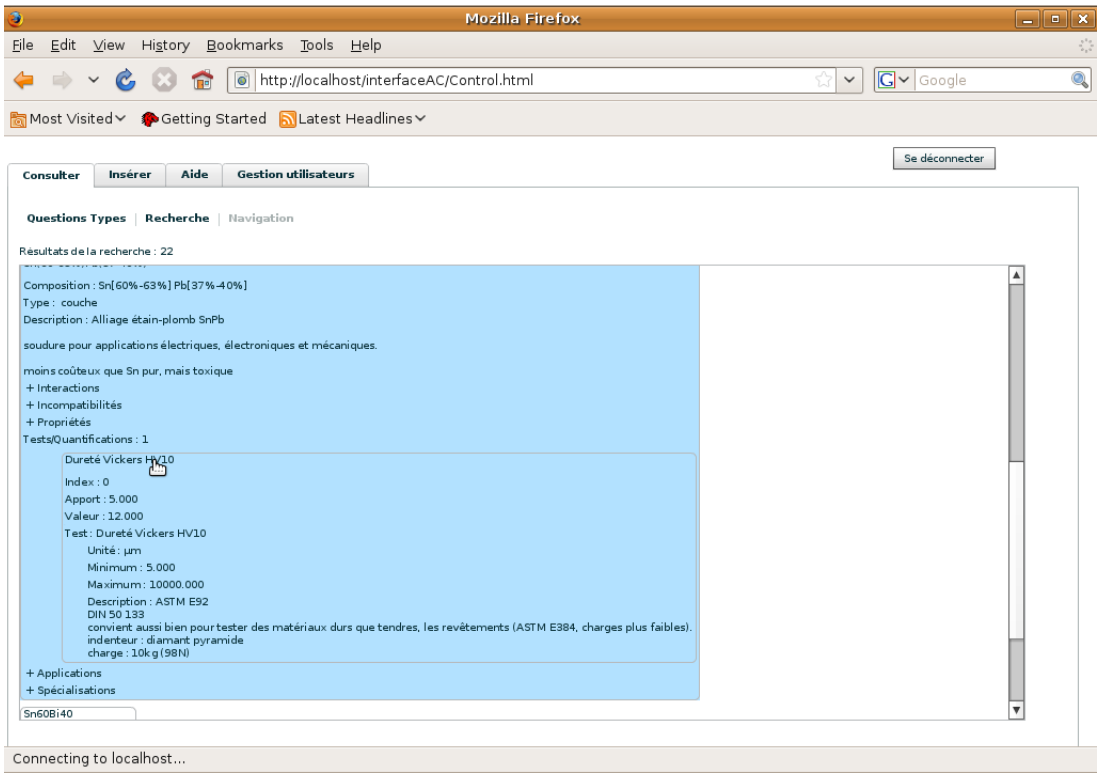


FIG. 4.22 – Déploiement d'un résultat de type matériau

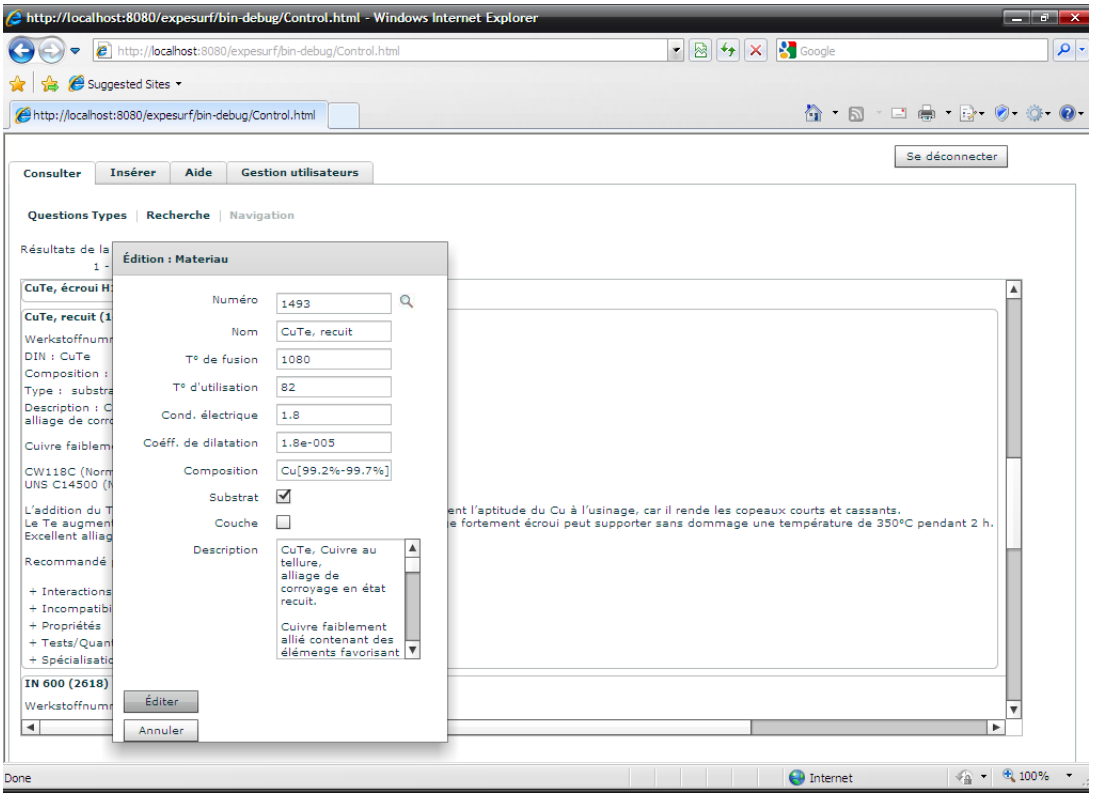


FIG. 4.23 – Edition

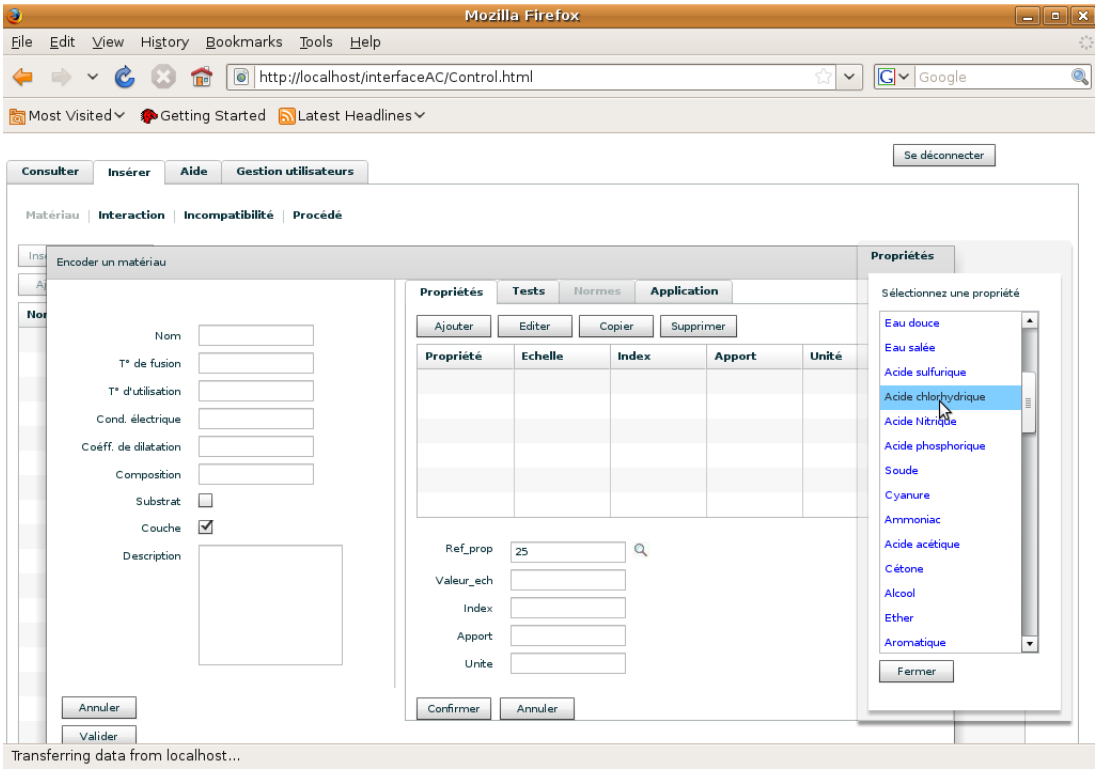


FIG. 4.24 – Trouver une propriété pour compléter un formulaire

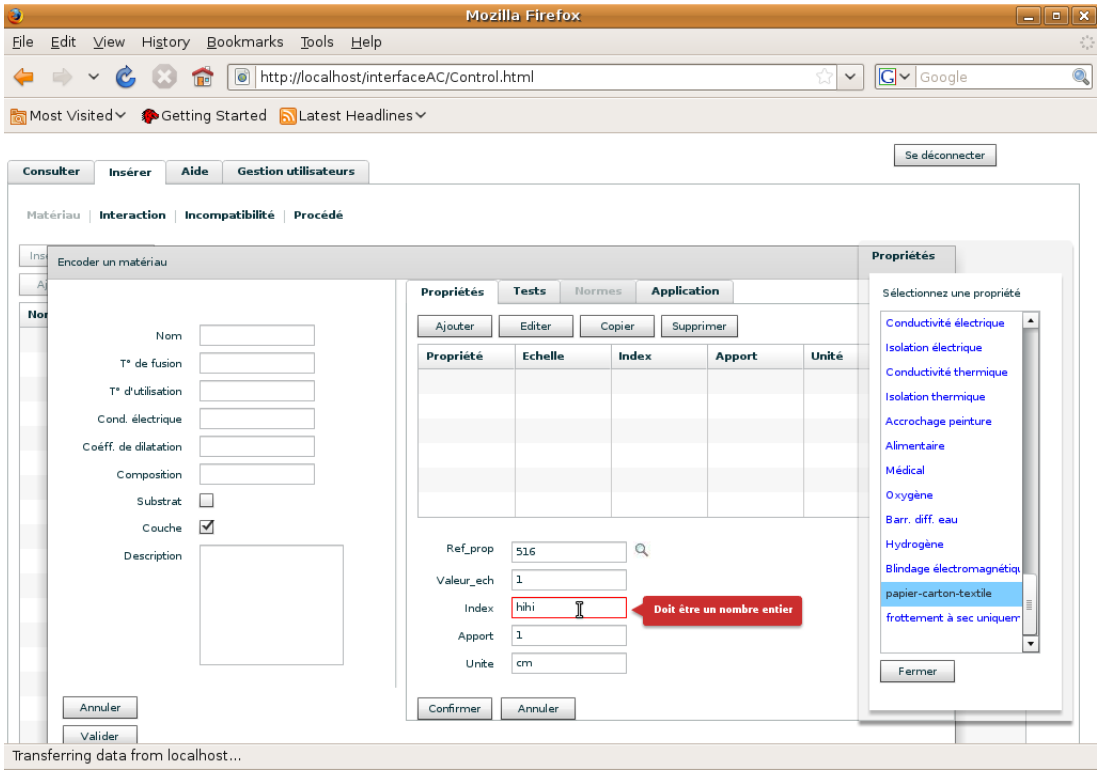


FIG. 4.25 – Correction d'un formulaire

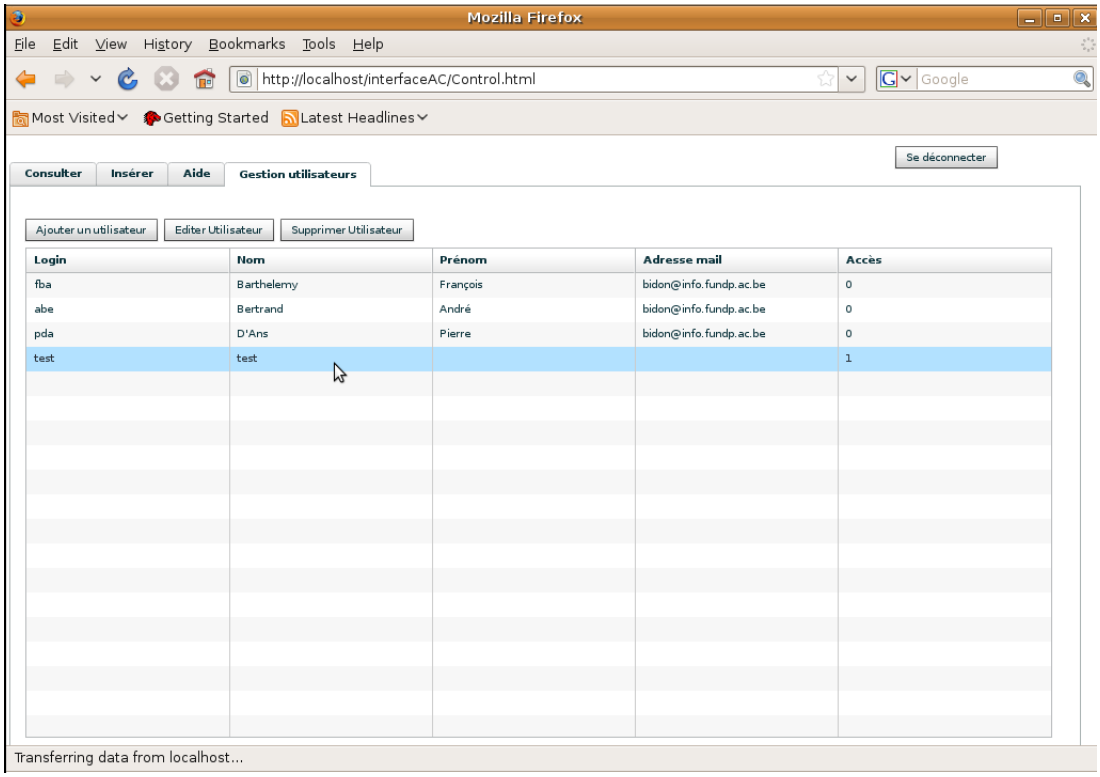


FIG. 4.26 – Gestion des utilisateurs

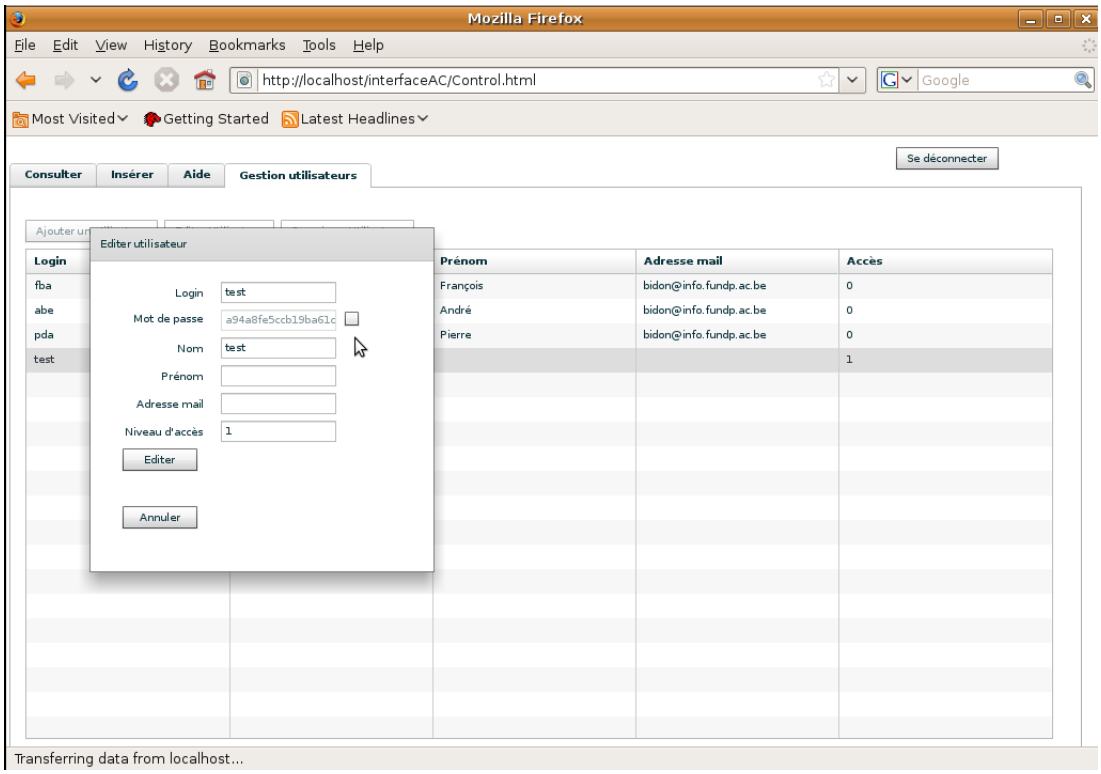


FIG. 4.27 – Formulaire d'édition d'un utilisateur

Chapitre 5

Travaux futurs

Au delà de la phase de test, certaines idées de développements futurs prennent forcément naissance dans le cadre d'un projet tel que celui-ci. Les différentes discussions avec le client permettent parfois de mettre en évidence des développements qui dépassent le cadre du mémoire ou qui ne seraient pas envisageables dans les délais requis. Il existe aussi les tâches qui sont déjà projetées et qui ne constituent pas uniquement des améliorations possibles, mais nécessaires. Nous avons distingué dans l'ensemble de ces idées les améliorations techniques des améliorations fonctionnelles de l'interface.

5.1 Améliorations techniques

Les améliorations techniques n'augmenteront pas l'ensemble des fonctionnalités de l'interface mais la feront gagner en sécurité, en performance ou encore, en confort d'utilisation.

S'affranchir de Zope pour l'interface d'acquisition de connaissances était un premier pas vers la suppression de Zope dans son ensemble. En effet, actuellement, la session d'utilisateur de l'interface d'acquisition de données dépend encore du système d'authentification et de session de Zope. La nouvelle interface développée utilise un système de session qui devrait être facilement adaptable à l'interface d'acquisition de données. Ainsi, l'objectif serait de fusionner le système d'authentification des deux interfaces.

Dans ce cas, il serait intéressant de veiller également à améliorer le système de session mis en place. La session étant vouée à être modifiée et la sécurité de l'interface n'étant pas la première priorité du nouveau développement elle est sommaire. Il serait intéressant de faire en sorte que les sessions soient davantage sécurisées ou qu'elles expirent éventuellement au bout d'un certain temps.

Une légère optimisation à laquelle il serait intéressant de songer est la désactivation des triggers. Ces triggers serviront tant que l'interface Zope sera toujours en service. Ils serviront également si l'on décide un jour de réinstaurer un système de contre-validation. Il n'est donc pas intéressant de les supprimer. En revanche, si on les désactive on peut supprimer les appels à la base de données pour les désactiver. Cela reviendrait à faire un appel à la base de données plutôt que trois pour chaque insertion.

Nous avons mentionné le fait de rassembler le système d'authentification et de session d'utilisateur pour les deux interfaces... A l'avenir, il serait intéressant d'aller au-delà et de rassembler les deux interfaces (celles d'acquisition de données et celle d'acquisition de connaissances). En effet, grâce aux deux niveaux d'administration de la nouvelle interface nous avons les moyens d'autoriser le parcours des données sans en autoriser l'insertion. Il serait intéressant pour l'utilisateur commun d'avoir donc un seul site pour soumettre ses problèmes au système et pour parcourir l'ensemble des données de la base de données. De même, l'utilisateur administrateur pourrait souhaiter effectuer de la consultation en même temps que de consulter les résultats des problèmes soumis.

A court terme l'ensemble des utilisateurs de l'interface est connu et limité. A plus long terme il est possible que des questions d'accessibilité puissent se poser et l'application devrait être adaptée afin qu'elle puisse être utilisée par des personnes malvoyantes ou non-voyantes. Flex 4 a pris soint de définir certains composants pour qu'ils puissent s'adapter à des logiciels pour malvoyants ou adapter leur format. Tous les composants utilisés dans l'application peuvent être adaptés pour être rendus accessibles même si ce n'est actuellement pas le cas. Bien qu'il ne s'agisse pas d'une application grand public et que les chances d'avoir un utilisateur de ce type sont minimes, c'est une adaptation qui valoriserait l'application tout en

ne réclamant pas de grosses modifications.

L'échéance de ce développement n'a pas permis non plus de mettre en oeuvre une bonne gestion des exceptions. La gestion est basique mais ne permet pas de discriminer entre les différents types d'erreurs qui surviennent. De même, lorsqu'une insertion échoue auprès de la base de données, à moins d'une erreur de connexion, il n'y a pas de discrimination entre les différents types d'erreurs SQL. Ainsi, l'utilisateur est prévenu que son encodage n'a pu être inséré mais n'a pas de feedback quant aux raisons de cet échec : deux fois le même identifiant etc. Si un contrôle important du remplissage des données à l'encodage permet de contourner largement ce problème il serait intéressant à l'avenir de renvoyer depuis le serveur une information pertinente décrivant l'origine de l'erreur.

5.2 Améliorations fonctionnelles

La période de développement et de test ont éveillé l'imagination des clients qui prennent tout doucement conscience de toutes sortes de petite améliorations qui faciliteraient leur quotidien. On se rend mieux compte une fois que le logiciel commence à prendre forme des possibilités fantastiques qu'il y aurait moyen de mettre en place pour rendre l'application beaucoup plus puissante.

Parmi les améliorations moins pressantes mais importantes à plus long terme on compte la possibilité d'effectuer des insertions pour toutes les tables. Actuellement, les insertions requises ont été limitées aux tables où elles étaient les plus fréquentes et à celles où il est impossible d'effectuer l'insertion à même la base de données (à cause de la double validation imposée). Or, il est inenvisageable que l'administrateur du système doive effectuer ses insertions manuellement au travers de l'interface de la base de données plutôt que de celle d'Expesurf. Ces insertions bien que moins fréquentes doivent donc être possible.

Il en va de même pour les recherches. On ne peut, actuellement, pas faire porter une recherche sur n'importe quel élément de la base de données mais que sur les éléments les plus importants. Il semble essentiel de travailler vers un système où toute information que l'on pourrait trouver dans la base de données soit recherchable au travers de l'interface afin de s'affranchir définitivement de l'interface de la base.

Parmi les fonctionnalités supplémentaires que l'on pourrait souhaiter de l'interface, la possibilité d'exporter et d'importer des encodages serait intéressante. On pourrait alors envisager que l'utilisateur effectue un certain nombre d'encodages et qu'il exporte ceux-ci avant de les insérer dans la base de données. Il pourrait manipuler les encodages en dehors de l'interface et les réimporter par la suite pour les enregistrer. Cela permettrait à l'utilisateur de travailler sans l'interface d'acquisition de connaissances et sans connexion au réseau.

De la même manière on pourrait envisager l'impression et l'exportation de fiches sous format de tableur à partir des résultats de recherche. On aurait ainsi accès aux informations de la base de données dans un autre format. On pourrait même réutiliser ces résultats afin de les modifier pour effectuer de nouveaux encodages. Après une première analyse, il semblerait que l'impression ne soit pas le point fort de Flex. Néanmoins, comme solution alternative, dans un premier temps, l'exportation des résultats en excel qui permettrait ensuite de les imprimer présenterait certainement un résultat plus propre et plus structuré.

5.3 Conclusions

Si certaines de ces modifications sont moins essentielles que d'autres, elles témoignent toutes du potentiel de l'application dans son ensemble. L'utilisation du langage Flex en particulier présente d'innombrables possibilités de présentation des informations, de transformation de la présentation de certains composants afin qu'ils s'adaptent à de nouvelles exigences ou à des structures de données nouvelles. De nombreuses librairies en Flex, parfois payantes, proposent déjà des solutions avancées de présentations de données aux relations complexes. On peut très bien imaginer maintenant que la récupération des données est mise en place greffer sur l'interface d'autres modes de présentation de celles-ci par la suite. En conclusion, l'interface actuelle est développée dans une technologie qui lui laisse encore de nombreuses possibilités d'expansion.

Chapitre 6

Conclusion

Le système expert Expesurf nécessitera encore certainement de nombreux développements avant de pouvoir servir ses objectifs. Néanmoins, un jour ; il permettra aux entreprises wallonnes de résoudre plus aisément leurs problèmes de traitements de surfaces. Ce mémoire constitue un premier pas dans cette direction.

En effet, l'nterface d'acquisition de connaissances n'apportait plus satisfaction pour toutes sortes de raisons et ce mémoire avait pour objectif d'y remédier. Dans un premier temps, afin d'identifier ses faiblesses, et d'éventuelles solutions, nous avons effectué une rétro-ingénieurie de cette interface et de la base de données du système. Le bilan fût sans équivoque : l'interface devait faire l'objet d'un tout nouveau développement. Certains problèmes pratiques auraient probablement pu être facilement corrigés. Cependant, d'autres dysfonctionnements plus graves (portabilité, technologie obsolète), ne pouvaient être solutionnés qu'en repartant sur d'autres bases avec un nouveau développement.

Une première étape de ce développement visait à définir les exigences de la nouvelle interface. tout en tirant profit des leçons du premier développement. Dans un deuxième temps, nous avons fait le choix d'une technologie en nous assurant d'éviter les erreurs passées. Finalement, après avoir tranché face à différents choix de développements et construit l'architecture de notre logiciel, nous avons découvert le framework Flex.

Soumis à une première phase de test, le prototype répond positivement aux comportements attendus. Il permet, d'une part, l'insertion, la suppression, et l'édition d'un ensemble de composants définis comme prioritaires. La nouvelle interface propose également un nouvel outil de visualisation et de navigation dans les données. Ce nouveau mode de consultation permet déjà de mettre en évidence des relations et enregistrements problématiques qui pourraient expliquer certaines solutions inadéquates suggérées par le système expert.

Cette nouvelle interface n'est pas un produit figé et définitif.Comme tout logiciel, il a une vie qui s'étend bien au delà se son premier développement. C'est pourquoi, il sera essentiel de s'assurer que l'application aura les moyens de se développer et de continuer à s'adapter aux besoins de ses utilisateurs.

Dans un premier temps il faut mener à terme la phase de test et les corrections de la nouvelle interface. Une fois cette phase terminée, certaines modifications impératives s'imposent. L'application est incomplète dans la mesure où elle ne permet pas de se passer complètement de l'interface d'administration de la base de données pour toutes les opérations d'édition, de suppression, et d'insertion. Ensuite, le système de session d'utilisateur est sommaire. Il devrait être amélioré et sécurisé afin d'être utilisé conjointement avec l'interface d'acquisition de données. Ces modifications sont moins importantes qu'il n'y paraît. L'architecture du logiciel a été conçue pour accueillir facilement ces nouveaux développements. Les fonctionnalités proposées sont implémentées complètement. Le futur développeur ne complétera donc pas une implémentation mais développera, à l'aide de la structure en place, les nouvelles fonctionnalités sans que cela n'altère le fonctionnement de l'interface actuelle.

Ces premières modifications me semblent être prioritaires. Toutefois, d'autres types de fonctionnalités seraient intéressantes à proposer : l'impression, l'exportation de fiches, l'importation d'encodages... Ces fonctionnalités, elles aussi, peuvent être facilement intégrables à l'architecture existante. De plus, l'utilisation de Flex offre d'emblée un grand nombre de librairies de qualité et un soutien lié à une communauté active d'utilisateurs.

Toutes les précautions prises par le développeur ne pourront jamais garantir que l'application rencon-

trera exactement les attentes des futurs utilisateurs, ni qu'elle ne subira pas le même sort que l'interface précédente. Cependant, une démarche rigoureuse et une documentation précise permettront d'appréhender avec sérénité les développements futurs.

Partant de l'ancienne interface qui posait problème tant du point de vue de ses fonctionnalités que de la technologie dans laquelle elle avait été développée, ce mémoire a mené à un nouveau développement qui répond à la fois aux nouvelles exigences des utilisateurs, corrige les erreurs de l'interface précédente et fait usage d'une technologie prometteuse pour les développements futurs.

Glossaire

action listener	Un action listener est le terme employé pour décrire un type d'objet qui exécute une procédure donnée lorsqu'un événement déterminé a lieu., 66
contrainte référentielle	La définition d'une contrainte référentielle définit une condition d'intégrité qui doit être satisfaite par toutes les lignes dans deux tables. La dépendance résultante affecte leurs enregistrements., 76
IDE	Integrated development environment. Outil facilitant le développement d'applications dans un langage particulier en assistant l'écriture, la corrections d'erreurs, et la compilation entreautres., 62
injection SQL	C'est une technique d'injection de code qui exploite une faille de sécurité au niveau du langage appellant le sql. Cela ne peut se produire que lorsqu'on a un code SQL imbriqué dans un autre code (code d'un corps de programme utilisant la base de données). Le code injecté dans une préparation peu sécurisée de la requête peut s'exécuter inopinément et avoir des effets pervers sur l'exécution du programme et sur la base de données., 63
interpréteur	Un interpreteur est un programme qui exécute un morceau de code écrit dans un langage de programmation déterminé. Il peut faire cela en éxécutant le code directement, en exécutant un code précompilé par un compilateur, ou en traduisant le code dans un langage intermédiaire plus efficace qu'il exécute ensuite., 59
langage orienté objet	On dit d'un langage de programmation qu'il est orienté objet à partir du moment où il est conçu de manière à faciliter la programmation orientée objet. Il s'agit d'un paradigme de programmation où les briques logicielles du programme sont vues comme des objets capables d'interragir entre eux. Les objets peuvent représenter une idée ou un élément du domaine d'application. Il possède un ensemble d'attributs qui le caractérisent et un ensemble de fonctions qui définissent son comportement., 42

middleware
: Un middleware est une couche de logiciels qui lie différents composants logiciels. Il consiste en un ensemble de services qui permettent à plusieurs procesus d'interragir au travers d'un mode de communication prédéfini., 63

model - view - controller
: Il s'agit d'un design pattern où l'on sépare les opérations de mise en page, des opérations d'accès aux informations, des traitements. Description plus complète par la suite., 48

moteur d'inférence
: Un moteur d'inférence est un composant logiciel qui essaie de développer des réponses sur base d'une base de connaissances., 11

PDO
: Les PHP Data objects constituent une extension du langage PHP qui définit une interface d'accès aux bases de données. Certains traitements sont optimisés (récupérations des données de la table) et certaines opérations sécurisées., 63

requête préparée
: Requête auprès d'une base de données qui fait l'objet d'une préparation permettant d'en optimiser l'exécution. Le temps de préparation rend l'utilisation de requêtes préparées plus ou moins intéressantes en fonction du nombre d'exécutions que l'on compte en faire. Ce type de requête peut être paramétré. Dans ce cas, certaines librairies effectuent une vérification de type et d'injection SQL, 63

schéma logique
: Un schéma logique est un modèle de données spécifique au domaine qu'il représente mais pas à la technologie dans laquelle il est/peut être implémenté., 17

SDK
: *Software development kit :* Ensemble d'outils de développement conçus pour aider le développement d'applications dans un langage ou environnement spécifique. Souvent accompagnés d'exemples, de documentation, et présentés dans un IDE (voir IDE), ils sont proposés par les résponsables de l'environnement., 62

trigger
: En base de données, un trigger est un code procédural exécuté automatiquement en réponse à certains événements sur une table ou une vue., 25

vue
: En bases de données, une vue consiste en une requête enregistrée dont les résultats sont consultables comme s'ils constituaient une table. Il s'agit d'une table virtuelle dynamique qui ne fait pas partie du schéma physique de la base de données., 28

WYSIWYG — *What you see is what you get :* l'acronyme désigne les types d'éditeurs qui présentent l'information non pas telle qu'elle est enregistrées mais telle qu'elle sera présentée après un traitement particulier. L'éditeur de texte Microsof Word est dit WYSIWYG parce qu'il cache à l'utilisateur les caractères de mise en page afin de présenter le document tel qu'il serait s'il venait à être imprimé., 62

Index

Bibliographie

[1] Programmer's Reference Guide to Zend AMF. `http://framework.zend.com/manual`.

[2] David Benyon, Phil Turner, and Susan Turner. *Designing Interactive Systems*. Addison Wesley, 2005.

[3] Millward Brown. Worldwide Ubiquity of Adobe Flash Player by Version. *Adobe.com*, June 2010.

[4] Fabien Deshayes. Les technologies riches. *Developpez.com, Club des développeurs*, 2007.

[5] F.Paternò. Model-Based Design and Evaluation of Interactive Applications. Tutorial at HCI 2000, 2000.

[6] Anthony Franco. Flex vs. Ajax, Friends or foes. *UI Resource center* *http://www.uiresourcecenter.com/ui-technologies/adobe-flex/whitepapers/FlexVsAJAXFriendsOrFoes.pdf* *(dernier accès le 27 Août 2010)*, Janvier 2008.

[7] Florent Garin. JavaFx vs Flex vs Silverlight. *florentgarin.org*, 2009.

[8] Jean-Luc Hainaut. *Bases de données et modèles de calcul*. Dunod, 2005.

[9] Amos Latteier, Michel Pelletier, Chris McDonough, Evan Simpson, Tom Deprez, Paul Everitt, Bakhtiar A. Hamid, Geir Baekholt, Thomas Reulbach, Paul Winkler, Peter Sabaini, Andrew Veitch, Kevin Carlson, Joel Burton, John DeStefano, Tres Seaver, Hanno Schlicting, and the Zope Community. The Zope2 Book. *Zope.org* *http://docs.zope.org/zope2/zope2book/* *(dernier accès le 27 Août 2010)*, 2002.

[10] Duane Nickull. How Truly Open is Flash ? do we need open flash ? `http://technoracle.blogspot.com/2007/01/how-truly-open-is-flash-do-we-need-open_03.html`, Janvier 2007.

[11] Trygve Reenskaug. Models - Views - Controllers. Xerox Palo Alto Research Lab Report, Décembre 1979.

[12] Trygve Reenskaug. Thing, Model, View, Editor - an example from a planning system. Xerox Palo Alto Research Lab Report, Mai 1979.

[13] Wil Sinclair. Zend Framework : Zend_Amf Component Proposal. *Framework.zend.com*, Janvier 2008.

[14] Ryan Stewart. Ajax Or Flex ? how to select rich internet application technologies. `http://www.zdnet.com/blog/stewart/ajax-or-flex-how-to-select-rich-internet-application-technologies/216` (dernier accès le 27 Août 2010), Janvier 2007.

[15] ULB and FUNDP. Expesurf - Rapport technique annuel n°3, 2007.

[16] Erik Wurzer. Why should you be using PHPś PDO for Database Access. *Nettuts.com* *http://net.tutsplus.com/tutorials/php/why-you-should-be-using-phps-pdo-for-database-access/* *(dernier accès le 27 Août 2010)*, Mai 2010.

Table des figures

Annexe A

Modifications apportées à la base de données

A.1 Modifications apportées aux tables

Deux tables ont été créées et une troisième modifiée. La table utilisateur comprend maintenant un champ mot de passe. Les deux autres tables servent à l'enregistrement des recherches pré-enregistrées (`question`) et la seconde aux rubriques de l'aide aux utilisateurs (`aide`).

```
1  CREATE TABLE aide (
2      id integer DEFAULT nextval('materiau_id_mat_seq'::regclass) NOT NULL,
3      rubrique character varying,
4      titre character varying,
5      contenu text,
6      admin integer
7  );
8
9  CREATE TABLE utilisateur (
10     id_utilisateur integer NOT NULL,
11     nom character varying(120) NOT NULL,
12     prenom character varying(120) NOT NULL,
13     login character varying(20) NOT NULL,
14     mail character varying(120) NOT NULL,
15     mdpasse character varying,
16     admin integer
17 );
18
19 CREATE TABLE question (
20     id integer DEFAULT nextval('materiau_id_mat_seq'::regclass) NOT NULL,
21     description character varying,
22     nom_table character varying,
23     conditions character varying
24 );
```

A.2 Deux nouvelles fonctions

La base de données comprend maintenant deux nouvelles fonctions. La première permet de désactiver l'ensemble des triggers. La seconde de réactiver tous les triggers désactivés par la première.

disable_state_changes()

```
1  BEGIN
2
3  ALTER TABLE materiau disable TRIGGER all;
```

```
4  ALTER TABLE application disable TRIGGER all;
5  ALTER TABLE caracterisation disable TRIGGER all;
6  ALTER TABLE classification disable TRIGGER all;
7  ALTER TABLE critere disable TRIGGER all;
8  ALTER TABLE evaluation disable TRIGGER all;
9  ALTER TABLE incompatibilite disable TRIGGER all;
10 ALTER TABLE interaction disable TRIGGER all;
11 ALTER TABLE procede disable TRIGGER all;
12 ALTER TABLE prop_doubles disable TRIGGER all;
13 ALTER TABLE propriete disable TRIGGER all;
14 ALTER TABLE qualification disable TRIGGER all;
15 ALTER TABLE quantification disable TRIGGER all;
16 ALTER TABLE resolution disable TRIGGER all;
17 ALTER TABLE solution disable TRIGGER all;
18 ALTER TABLE specialisation disable TRIGGER all;
19 ALTER TABLE test disable TRIGGER all;
20
21 return;
22 END
```

enable_state_changes()

```
1  BEGIN
2
3  ALTER TABLE materiau enable TRIGGER all;
4  ALTER TABLE application enable TRIGGER all;
5  ALTER TABLE caracterisation enable TRIGGER all;
6  ALTER TABLE classification enable TRIGGER all;
7  ALTER TABLE critere enable TRIGGER all;
8  ALTER TABLE evaluation enable TRIGGER all;
9  ALTER TABLE incompatibilite enable TRIGGER all;
10 ALTER TABLE interaction enable TRIGGER all;
11 ALTER TABLE procede enable TRIGGER all;
12 ALTER TABLE prop_doubles enable TRIGGER all;
13 ALTER TABLE propriete enable TRIGGER all;
14 ALTER TABLE qualification enable TRIGGER all;
15 ALTER TABLE quantification enable TRIGGER all;
16 ALTER TABLE resolution enable TRIGGER all;
17 ALTER TABLE solution enable TRIGGER all;
18 ALTER TABLE specialisation enable TRIGGER all;
19 ALTER TABLE test enable TRIGGER all;
20
21 return;
22 END
```

Annexe B

Diagrammes de classes

B.1 Les diagrammes de classes du modèle UML

UML (Unified Modeling Language) est un langage de modélisation graphique qui propose plusieurs types de diagrammes. Nous présenterons dans ce chapitre le diagramme de classe, considéré comme l'élément central d'UML.

B.2 Diagrammes de classes du client de la nouvelle interface

Un diagramme de classe permet de réprésenter graphiquement la structure interne d'un domaine spécifique. Ainsi, ce type de diagramme se construit sur base de trois types d'éléments :

1. Des classes.
2. . Des attributs.
3. Des opérations.

Notons qu'un attribut et une opération sont toujours propre à une classe comme nous le verrons ci-après.

B.2.1 Représentation d'une classe

Une classe représente un tye d'entité du domaine. Elle possède un certain nombre d'attributs ainsi que d'éventuelles opérations. Ainsi, par exemple, imaginons une classe "chien". Un chien peut être défini par sa race, sa couleur, son sexe et son nom. Un chien peut boire, manger, dormir et jouer. Cette classe "chien" est représentée à la figure

La classe chien possède ici 4 attributs : race, couleur, sexe et nom. Un attribut est donc une donnée de la classe permettant de la caractériser. Notons qu'il existe la notion de visibilité d'un attribut qui comprend trois "niveaux" d'accès à cet attribut :

- l'accès public (représenté par un "+" dans le graphique) qui offre à toutes les autres classes l'accès à l'attribut.
- l'accès protégé (représenté par un "#") qui limite l'accès à la classe elle-même et ses enfants.
- l'accès privé (représenté par un "-") qui limite l'accès à la classe elle-même uniquement.

De plus, un attribut est une donnée qui sera représentée selon un type bien précis qu'il faut préciser. Ainsi comme nous voyons, l'attribut race sera de type "String".

Dans notre exemple, la classe chien possèdes 4 opérations :boire, manger, dormir et jouer. La notion de visibilité s'applique également aux opérations. De plus, une opération se présente comme une fonction informatique classique. Celle-ci peut prendre éventuellement des paramètres en entrée et retourner une valeur.

Ainsi, dans notre exemple, l'opération " + manger(repas :nourriture) :void" possède un paramètre d'entrée : "repas" qui est de type "nourriture" A contrario, l'opération "+dormir()" ne prend aucun paramètre d'entrée.

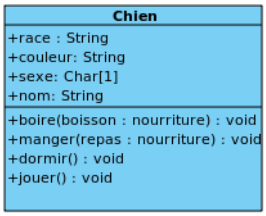


FIG. B.1 – Diagramme de classe chien

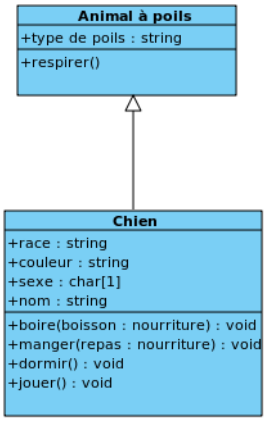


FIG. B.2 – Diagramme de classe : héritage

B.2.2 Relation entre classes

Il existe plusieurs relations possibles entres classes : l'héritage, l'association, l'agrégation et la composition. Dans le cadre de ce mémoire, seules les relations d'héritage et d'association ont été utilisées. Dans le but de ne pas embrouiller inutilement le lecteur, seules les deux relations utilisées seront décrites ci-après.

B.2.2.1 L'héritage

Comme son nom l'indique, cette relation permet de définir la ou les classes qui "héritent" d'une autre. Ainsi, si nous reprenons notre exemple de classe "chien". Imaginons une classe "Animal à poils". Cette classe se définit par une série d'attributs et d'opérations qui lui sont propres comme par exemple l'attribut "type de poils" et l'opération "respirer". Ici clairement, la classe "chien" hérite de la classe "animal à poils". Ainsi, la classe "chien" hérite des attributs et opérations "d'animal à poils" et donc possèdera également l'attribut "type de poils" et l'opération "respirer". On dira dans ce cas que la classe chient "spécialise" la classe "animal à poils". Dans le sens inverse, on parlera de "généralisation". La représentation graphique de cet héritage se trouve à la figure

B.2.2.2 L'association

Il s'agit d'une relation sémantique entre les objets d'un classe représenté par un trait plein entre les classes associées. Le rôle indique par chaque intervenant dans la relation peut être indiqué. Il existe également la notion de multiplicité qui permet de préciser le nombre minimum et maximum d'instances de chaque classe qui sont concerné par la relation. Ainsi par exemple, si nous imaginons une classe "famille" et une relation entre notre classe "chien" et cette nouvelle classe. Nous pourrions décrire cette relation comme le fait qu'une famille possède ou non un chien. Ainsi, un chien peut ou non appartenir à une famille. A l'inverse, une famille pourrait posséder de 0 à plusieurs chiens. Cette exemple d'association est représenté à la figure

Le langage UML nous permet donc de représenter sous un formalisme graphique clair et précis l'ensemble des entités composant notre domaine de recherche.

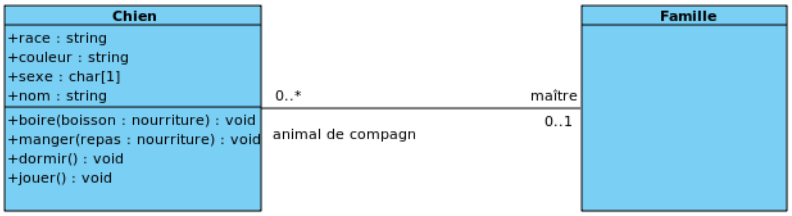


FIG. B.3 – Diagramme de classe : association

B.2.2.3 Les classes impliquées dans l'insertion d'intéractions et d'incompatibilités

Selon la même logique que celle utilisée pour l'insertion de matériaux la structure s'organise encore une fois ici entre autour d'une classe centrale : InterView et/ou IncompView (voir figure B.4). Elles proposent la présentation et la coordination de l'enregistrement et l'insertion d'encodages d'interactions et d'incompatibilités respectivement. Pour ce faire elles utilisent des classes d'objet visant à modulariser la présentation et représentant une abstraction de formulaires d'insertion comme on en a déjà vu préalablement. Et ensuite, elles font usage d'une classe dont une instance joue le rôle de modèle de données en accédant à et en modifiant les données auprès du serveur.

B.2.2.4 Les classes impliquées dans l'insertion de procédés

Nous retrouvons encore exactement la même construction au niveau de l'insertion des procédés à la figure B.5. Le modèle est ici particulièrement complet. En effet, chacun de ces modèles est forcément utilisé à deux fins différentes : l'insertion et également la recherche. Dans le cadre de la recherche, la présentation des résultats est gérée par la classe Control et en particulier la méthode rendererChooser qui est appellée par specSearch lors d'une recherche. Lors de la présentation des résultats, différentes méthodes des modèles seront utilisées afin de se procurer les informations complémentaires.

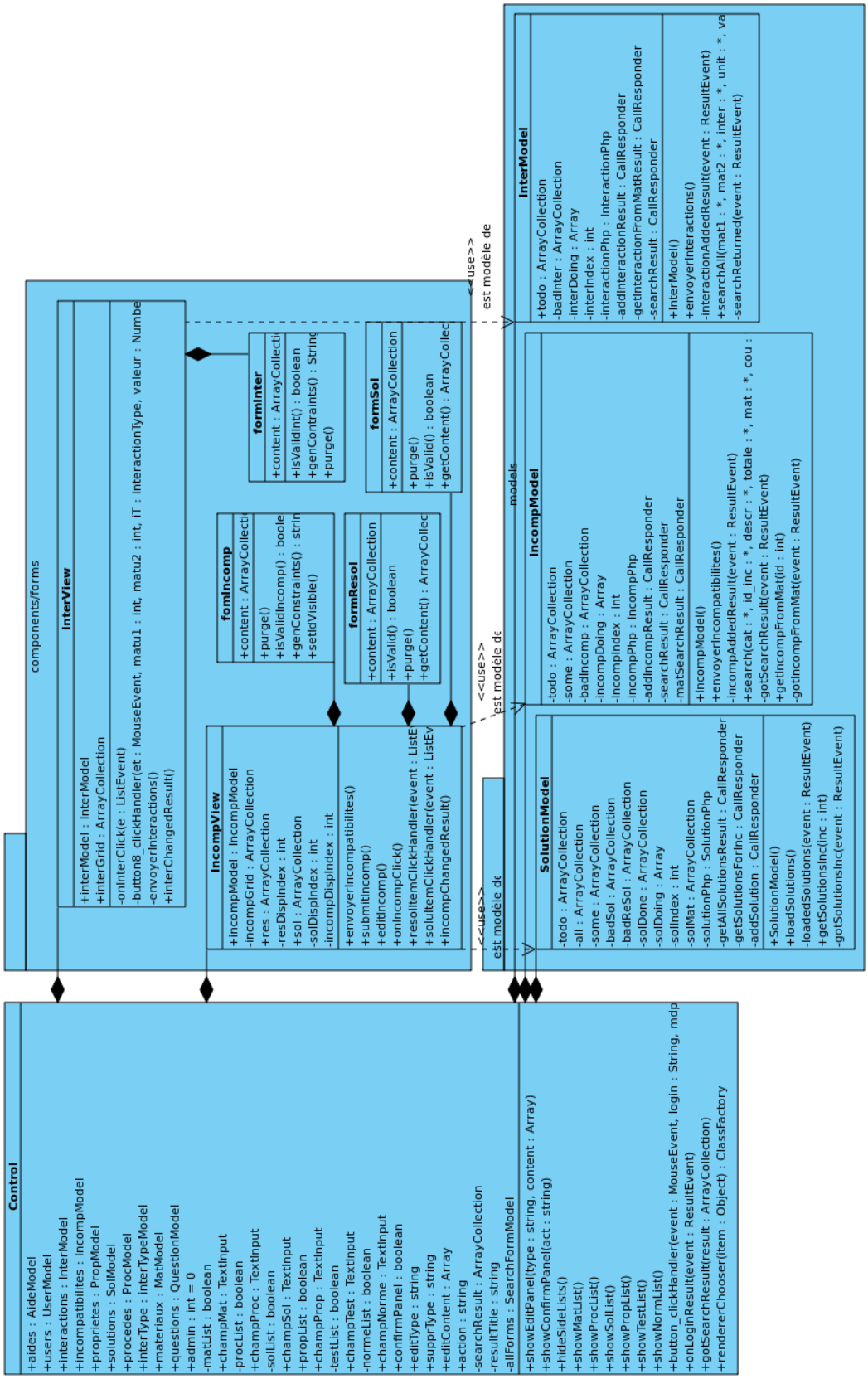


FIG. B.4 – Diagramme de classe de l'insertion d'intéractions et d'incompatibilités

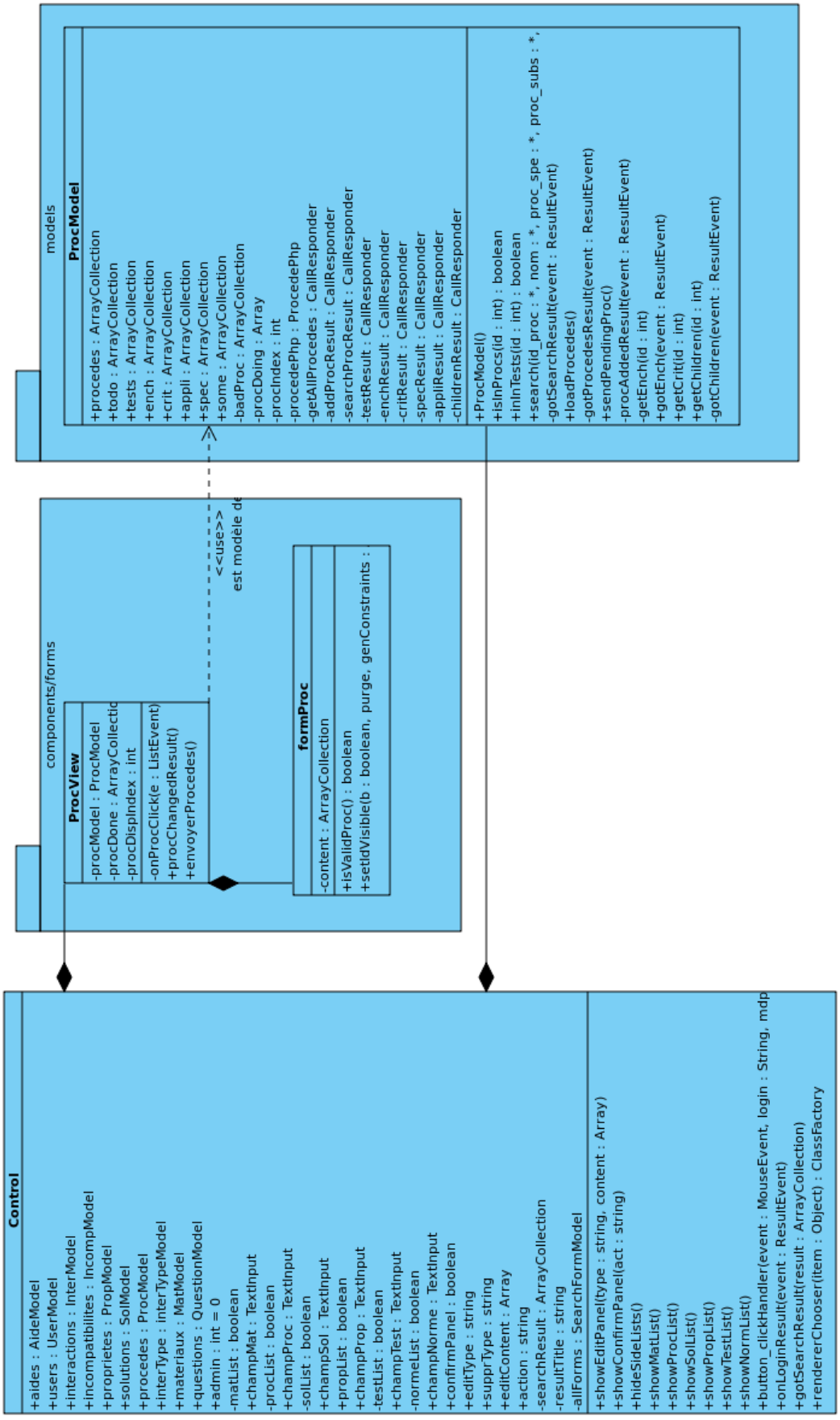


FIG. B.5 – Diagramme de classe de l'insertion de propriétés

Sandy Wauthier

Annexe C

La machine virtuelle

Ce mémoire vient accompagné d'un DVD qui permet de tester la nouvelle interface et de découvrir également les autres interfaces et l'ensemble du système une fois installé. En effet, le DVD contient les fichiers d'installation d'une machine virtuelle tournant avec le logiciel VirtualBox. Cette annexe propose une procédure d'installation de la machine virtuelle et la présente ensuite.

C.1 Installation de la machine virtuelle

L'installation de la machine virtuelle est très simple mais peut prendre un certain temps.[1]

Installation de VirtualBox : Dépendant du système d'exploitation sous lequel vous travaillez, le processus d'installation sera légèrement différent. Cependant, il n'est pas plus complex que l'installation de n'importe quel autre logiciel.

1. Le fichier d'installation peut être téléchargé à l'adresse `http://www.virtualbox.org/wiki/Downloads`[2].
2. Téléchargez le fichier d'installation correspondant à votre système d'exploitation et exécutez-le. L'installation n'implique aucune opération particulière. Il vous suffit de sélectionner «suivant» à chaque étape nécessaire et d'accepter le contrat de licence.
3. Une fois le logiciel installé, il apparaîtra soit dans vos nouveaux programmes (sous Windows) soit dans vos «accessoires» sous Ubuntu.

Importation de la machine virtuelle :

1. Enregistrer les fichiers *.OVF* et *.VDMK* du DVD sur votre ordinateur et démarrez VirtualBox.
2. Sélectionner «import appliance» dans le menu «File» (figure C.1).
3. Sélectionner «choose» et parcourir vos répertoires à la recherche du fichier *.OVF*. Sélectionner le fichier (figure C.2.
4. Sélectionner «suivant». Il vous présente alors les caractéristiques de la machine (figure C.3). Sélectionnez «import» (l'importation peut prendre un certain temps).
5. La nouvelle machine virtuelle apparaît dans la fenêtre principale de VirtualBox. Votre machine virtuelle est installée (figure C.1).

Utilisation :

1. Pour lancer la machine virtuelle double cliquez dessus dans la fenêtre d'accueil de VirtualBox (fenêtre de la figure C.1).
2. Pour rentrer dans la machine virtuelle il vous suffit de cliquer dedans. VirtualBox vous demande alors de choisir si vous souhaitez que la capture de la souris soit automatique.
3. Pour un plus grand confort visuel mettez la machine virtuelle en plein écran (dans le menu «machine»).
4. La touche *ctrl+dr* (ou *ctrl+alt+delete*) permet de quitter la machine virtuelle.

[1] La procédure d'installation est inspirée de celle disponible sur le site `http://www.ciresearchgroup.org/documentation/virtualbox-setup-instructions`

[2] L'installation sous Ubuntu peut se faire en ligne de commande en utilisant la commande `sudo apt-get install virtualbox virtualbox-ose-modules-generic` mais elle ne vous permet pas nécessairement de télécharger la dernière version du logiciel. Or, l'importation d'une machine virtuelle est une fonctionnalité récente du logiciel.

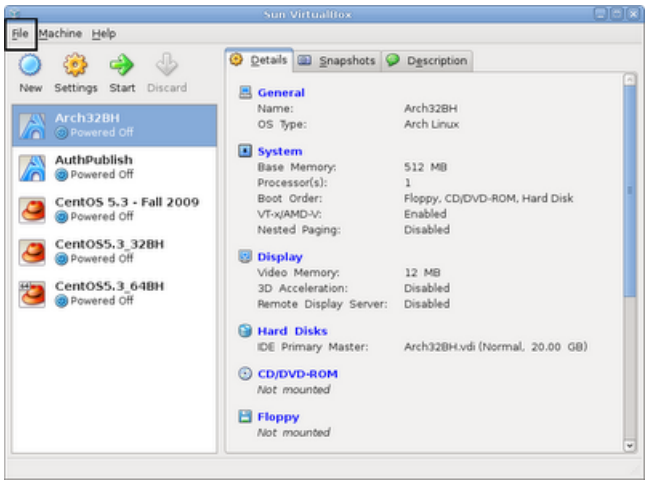


FIG. C.1 –

C.2 Description de la machine virtuelle

Ubuntu réclame une authentification de l'utilisateur pour se logguer. Voici ce compte utilisateur :

Utilisateur : «etd»

Mot de passe : «memoire»

Une fois correctement authentifié dans la machine virtuelle, vous pouvez lancer Firefox. La page de lancement propose les liens pour accéder aux différentes parties du logiciel. Elle permet également d'accéder au code source du logiciel et au petit exemple de développement Flex qui illustre le chapitre Développement.

Seule la nouvelle interface nécessite de compléter ses coordonnées manuellement (informations sur la page d'accueil de Firefox). Les login et mots de passe des autres interfaces ont été préenregistrées.

Le répertoire «Documents» comprend un document indiquant tous les comptes et mots de passes de la machine virtuelle ainsi que les chemins vers les différents répertoires de sources du système Expesurf.

L'installation sur la machine virtuelle permet, principalement, de découvrir les différents composants mais l'ensemble du système expert n'y est pas fonctionnel.

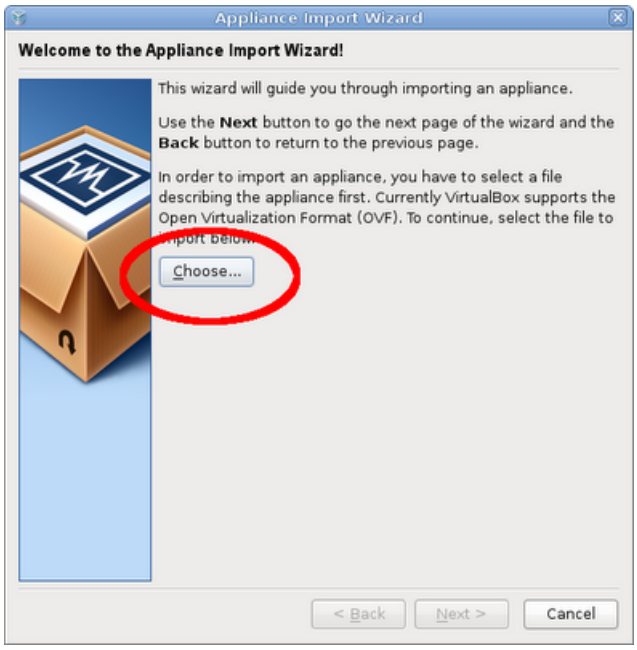

Appliance Import Wizard
Welcome to the Appliance Import Wizard!
This wizard will guide you through importing an appliance.
Use the Next button to go the next page of the wizard and the Back button to return to the previous page.
In order to import an appliance, you have to select a file describing the appliance first. Currently VirtualBox supports the Open Virtualization Format (OVF). To continue, select the file to import below.
Choose...
< Back
Next >
Cancel

FIG. C.2 –

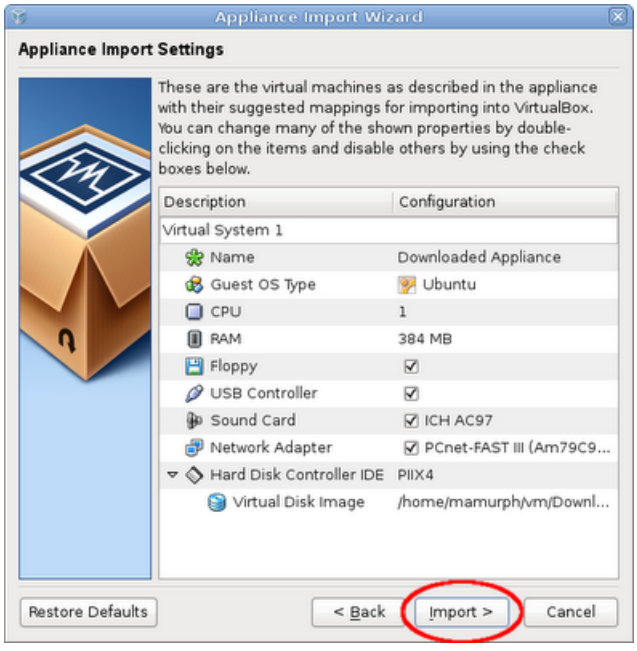

Appliance Import Wizard
Appliance Import Settings
These are the virtual machines as described in the appliance with their suggested mappings for importing into VirtualBox. You can change many of the shown properties by double-clicking on the items and disable others by using the check boxes below.
Description
Configuration
Virtual System 1
Name
Downloaded Appliance
Guest OS Type
Ubuntu
CPU
1
RAM
384 MB
Floppy
USB Controller
Sound Card
ICH AC97
Network Adapter
PCnet-FAST III (Am79C9...
Hard Disk Controller IDE
PIIX4
Virtual Disk Image
/home/mamurph/vm/Downl...
Restore Defaults
< Back
Import >
Cancel

FIG. C.3 –

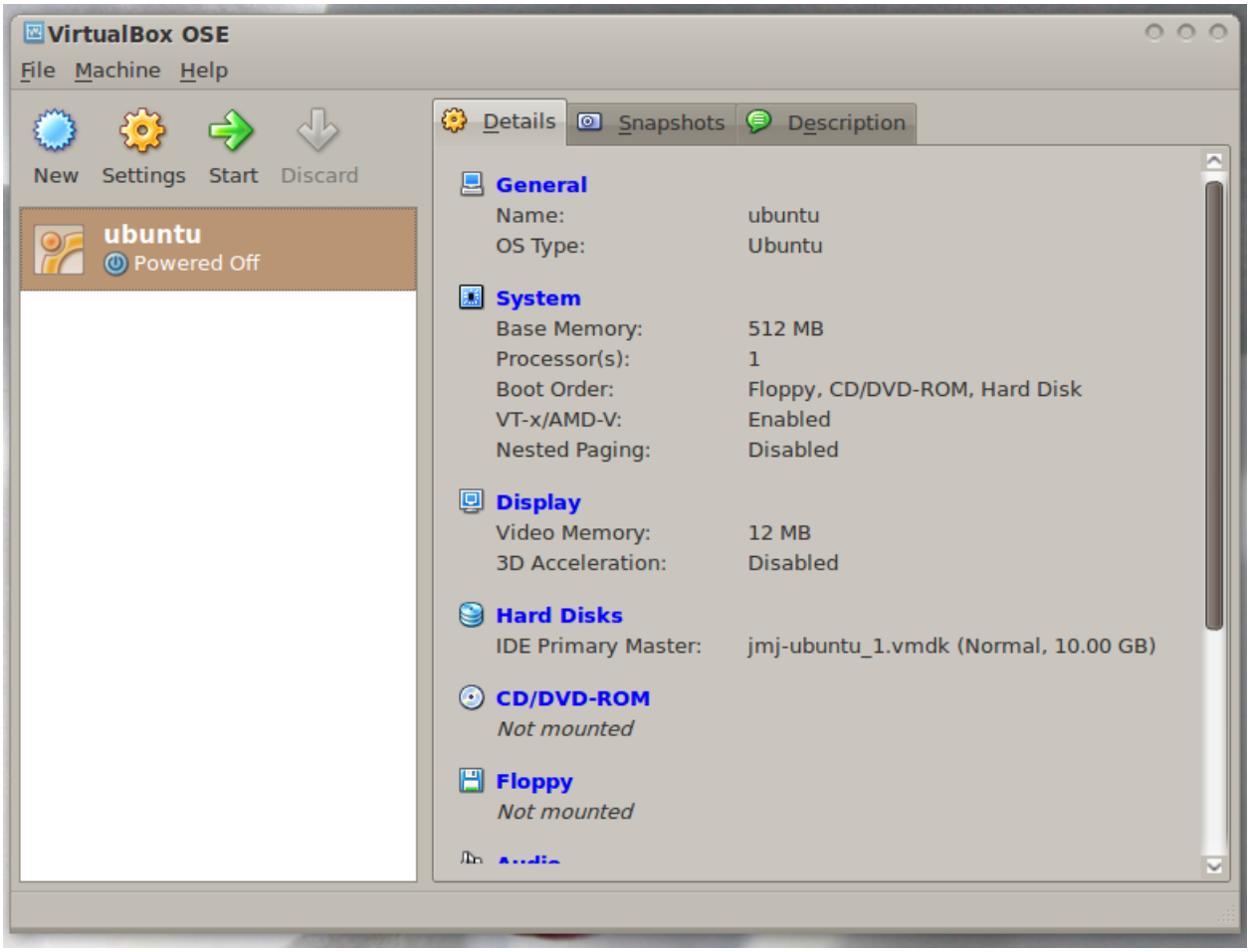

VirtualBox OSE
File Machine Help
New
Settings
Start
Discard
ubuntu
Powered Off
Details
Snapshots
Description
General
Name: ubuntu
OS Type: Ubuntu
System
Base Memory: 512 MB
Processor(s): 1
Boot Order: Floppy, CD/DVD-ROM, Hard Disk
VT-x/AMD-V: Enabled
Nested Paging: Disabled
Display
Video Memory: 12 MB
3D Acceleration: Disabled
Hard Disks
IDE Primary Master: jmj-ubuntu_1.vmdk (Normal, 10.00 GB)
CD/DVD-ROM
Not mounted
Floppy
Not mounted

FIG. C.4 –